Universidade Federal do Paraná

Rogério C. Turchetti

Construindo Sistemas Distribuídos Tolerantes a Falhas e Eficientes em Redes SDN com NFV

Curitiba PR
2017

Rogério C. Turchetti

Construindo Sistemas Distribuídos Tolerantes a Falhas e Eficientes em Redes SDN com NFV

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Informática no Programa de Pós-Graduação em Informática, setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Elias P. Duarte Jr.

Curitiba PR
2017

T932c Turchetti, Rogério C.
Construindo sistemas distribuídos tolerantes a falhas e eficientes em redes SDN com NFV / Rogério C. Turchetti. – Curitiba, 2017 .
120 f. : il. color. ; 30 cm.

Tese - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2017 .

Orientador: Elias P. Duarte Jr.

1. Ciência da computação. 2. Redes definidas por software. 3. Função virtualizada de rede. I. Universidade Federal do Paraná. II. Duarte Jr., Elias P. III. Título.

CDD: 005.4

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
Setor CIÊNCIAS EXATAS
Programa de Pós-Graduação INFORMÁTICA

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **ROGERIO CORREA TURCHETTI** intitulada: **Construindo Sistemas Distribuídos Tolerantes a Falhas e Eficientes em Redes SDN com NFV**, após terem inquirido o aluno e realizado a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de doutor está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

Curitiba, 15 de Agosto de 2017.

ELIAS PROCÓPIO DUARTE JUNIOR
Presidente da Banca Examinadora (UFPR)

LISANDRO ZAMBENEDETTI GRANVILLE
Avaliador Externo (UFRGS)

CARLOS ALBERTO MAZIERO
Avaliador Interno (UFPR)

ALCIDES CALSAVARA
Avaliador Externo (PUC/PR)

LUIZ ANTONIO RODRIGUES
Avaliador Externo (UNIOESTE)

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
Setor de Ciências Exatas
Coordenação de Pós-Graduação em Informática

Á minha esposa Patrícia e aos meus filhos Lorenzo e Bianca. Aos meus pais, Altair e Lúcia.

"Não há quem explique
e ninguém quem não
entenda."
Cecília Meirelles

Agradecimentos

Ao meu orientador e grande amigo, professor Elias, pelas palavras otimistas nos momentos mais intrincados, pelos longos períodos de orientação, sem nunca faltar com esforço para auxiliar nas resoluções dos problemas e principalmente pela confiança depositada em mim. Cada reunião serviu de inspiração contribuindo em muito para o desenvolvimento deste trabalho, a ele sempre serei grato.

Dedico este trabalho à minha esposa Patrícia e ao meu filho amado Lorenzo. Independente do lugar em que estivemos, nestas várias mudanças e momentos indecisos, sempre estiveram ao meu lado. Agradeço todos os dias por vocês fazerem parte de minha vida. Amor maior que tudo, "sem vocês, eu nada seria". Sem esquecer também de agradecer a Deus pelo anjinho que estás por vir. Já te amo muito, minha princesinha Bianca.

Meus mais sinceros agradecimentos a meu pai Altair que sempre foi o alicerce de nossa família e sempre nos manteve unidos, por mais que isso às vezes nos custasse várias lágrimas de despedidas. À minha mãe que amo tanto Lúcia, sempre me ensinou a seguir em frente e ter firmeza mesmo nos momentos de angústias. E à minha irmã Maritê que segurava as pontas quando o carro dobrava a esquina rumo à Curitiba, minha "mana" do coração. Agradeço também pela minha segunda mãe Geni, mulher guerreira e de fibra que nos ensina sempre a ter muita fé em Deus.

Durante os 4 anos de trabalho tive oportunidade de fazer inúmeras amizades. Agradeço aos meus amigos do LaRSis, em especial ao Giovanni, Edson e Luiz. A família Trois agradeço pelas confraternizações e pela parceria, vocês passaram a fazer parte de nossa família.

Por fim, agradeço aos meus colegas da UFSM, amigos que fazem parte deste projeto. E aos demais familiares que mesmo sem nossa presença, mantiveram nossa família sempre unida.

Resumo

Os sistemas de virtualização estão mudando a maneira de projetar e operar as redes de computadores. A Virtualização de Funções de Rede (NFV - *Network Function Virtualization*) é uma tecnologia emergente que implementa, utilizando técnicas de virtualização, funções de rede tradicionalmente fornecidas em dispositivos de hardware específicos. O presente trabalho tem por objetivo tirar proveito destas novas tecnologias para implementar, dentro da própria rede, componentes clássicos para a construção de sistemas tolerantes a falhas. A primeira parte do trabalho descreve implementações de serviços de detecção de falhas. A função virtualizada de rede NFV-FD (FD – *Failure Detector*) é executada em uma rede SDN (*Software Defined Network*), utilizando informações disponibilizadas por um controlador e por *switches* para monitorar o estado de processos e enlaces de comunicação. A NFV-FD foi implementada e seus benefícios são apresentados através de experimentos reportados no trabalho. Foi também implementado um serviço para detecção de falhas para a Internet (IFDS - *Internet Failure Detection Service*), que pode ser configurado de acordo com as necessidades de QoS (*Quality of Service*) das aplicações. Em particular, são propostas estratégias que permitem a configuração do IFDS tendo em vista os requisitos de múltiplas aplicações. O IFDS é composto por uma MIB (*Management Information Base*) SNMP (*Simple Network Management Protocol*) denominada $fdMIB$. Um protótipo do serviço foi implementado e resultados experimentais são apresentados, obtidos tanto em redes locais como na Internet. Outra contribuição desta tese relacionada aos detectores de falhas trata da questão do cômputo do *timeout*, sendo proposta uma estratégia denominada $tuning_{\phi}$ que reajusta o valor do *timeout* de acordo com os tempos de comunicação observados. Nos experimentos, $tuning_{\phi}$ apresentou um excelente desempenho, reduzindo de forma expressiva o número de falsas suspeitas. Em outra contribuição tratamos do problema da sincronização consistente entre múltiplos controladores SDN. Uma função virtualizada de rede denominada de *VNF-Consensus*, é proposta para garantir a sincronização consistente entre os controladores envolvidos. *VNF-Consensus* implementa o algoritmo de consenso Paxos e com sua utilização os controladores ficam desacoplados da sincronização. Os experimentos mostraram que utilizando a *VNF-Consensus*, o plano de controle é sincronizado sem aumentar a carga de trabalho nos controladores. Por último, propomos *AnyBone*, uma NFV que oferece as primitivas de difusão confiável para garantir a entrega ordenada das mensagens transmitidas na rede. O *AnyBone* é baseado em um sequenciador que gerencia as transmissões e entrega as mensagens ordenadas aos processos, além de oferecer uma API para as aplicações trocarem mensagens utilizando as primitivas de difusão atômica e confiável. Os resultados experimentais demonstram a eficiência da estratégia proposta, bem como seu custo, em termos da latência da difusão em diferentes cenários, ou seja, variando o número de participantes e o tamanho das mensagens transmitidas. O conjunto de contribuições desta tese permite concluir que é viável utilizar a própria rede para implementar com eficiência componentes clássicos de tolerância a falhas que podem ser disponibilizados como serviços para aplicações distribuídas diversas.

Palavras-chave: Função Virtualizada de Rede, Tolerância a Falhas, Sistemas Distribuídos, Redes Definidas por Software.

Abstract

Virtualization systems are changing the way networks are designed and deployed. Network Function Virtualization (NFV) is an emerging technology that employs virtualization to transform network devices into virtual entities. In this thesis, we take advantage of this technology to implement classical distributed systems abstractions within the network. As a result, we aim to be able to build efficient fault-tolerant distributed applications. Initially, we describe implementations of failure detectors. A virtual network function called NFV-FD (FD stands for - *Failure Detector*) is implemented in a Software Defined Network (SDN) and uses information obtained from a SDN controller to monitor processes and determine their state. In addition, NFV-FD also provides information about the state of communication links. NFV-FD was implemented and experimental results are reported. We also implemented an Internet Failure Detection Service (IFDS), which can be used to provide the Quality of Service (QoS) level required by the applications. In particular, we proposed two strategies to configure IFDS when multiple processes are monitored with different QoS requirements. IFDS was implemented with SNMP (Simple Network Management Protocol). We have implemented a prototype of the service and experimental results are presented running both on a single LAN and on the Internet. Another contribution of this thesis also related to failure detectors addresses the question of how to compute a precise timeout interval. We propose the $tuning_{\phi}$ strategy that dynamically adjusts the timeout interval in a way that better reflects a varying behavior of the communication channel. Experimental results obtained from running $tuning_{\phi}$ show that the strategy reduces significantly the number of false detections. The next contribution of the thesis refers to achieving consistent synchronization across multiple SDN controllers. In order to ensure consistent synchronization among controllers of a SDN distributed control plane, we propose the virtual network function *VNF-Consensus*. *VNF-Consensus* implements the Paxos consensus algorithm so that controllers are decoupled from synchronization tasks. Experimental results show that our solution is able to guarantee a consistent control plane without increasing the number of tasks a controller has to execute. Finally, we propose *AnyBone*, a VNF that offers reliable and atomic broadcast primitives, which ensures that messages are delivered by all the processes and in the same total order. *AnyBone* relies on a sequencer to manage the transmissions and enforce the order. Furthermore, *AnyBone* provides an API for applications to be able to employ atomic and reliable broadcast primitives. Experimental results show that *AnyBone* provides an efficient strategy to ensure the ordered message delivery to all processes. We measured the broadcast latency in different scenarios, i. e., increasing the number of processes involved in the communication and also the size of the transmitted messages. By taking into account the results of the contributions of this thesis, we can conclude that it is not only feasible, but also efficient to use the network itself in order to deploy classic fault tolerance abstractions which can be used to build fault-tolerant distributed applications.

Keywords: Network Function Virtualization, Fault Tolerance, Distributed Systems, Software Defined Networking.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

API	Application Programming Interface
BB	Broadcast-Broadcast
BSS	Business Support System
CAPEX	CAPital EXpenditures
DDoS	Distributed Denial of Service
EMS	Element Management System
ETSI	European Telecommunications Standards Institute
FD	Failure Detector
fdMIB	failure detector MIB
FDMod	Failure Detector Module
FIFO	First-In/First-Out
FPC	Fast Paxos-based Consensus
GCD	Greatest Common Divisor
IFDS	Internet Failure Detection Service
IPS	Intrusion Prevention System
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ISG	Industry Specification Group
LAN	Local Area Network
LLDP	Link Layer Discovery Protocol
MIB	Management Information Base
NAT	Network Address Translation
NFs	Network Functions
NFV	Network Function Virtualization
NFV-FD	Network Function Virtualization - Failure Detector
NFV-MANO	Network Functions Virtualization Management and Orchestration
NFVIaaS	Network Functions Virtualization Infrastructure as a Service
NFVO	Orchestrator
NIB	Network Information Base
NS	Network Services
OID	Object IDentifier
OPEX	OPerational EXpenditures
OSS	Operations Support System
OVS	Open vSwitch
PCE	Path Computation Element
QoS	Quality of Service
REST	REpresentational State Transfer
RFC	Request for Comments
RTO	Retransmission Timeout

RTT	Round Trip Time
SDC	Software-Defined Cloud
SDN	Software Defined Networks
SFC	Service Function Chains
SFP	Service Function Path
SLA	Service Level Agreements
SNMP	Simple Network Management Protocol
SOAP	Simple Object Access Protocol
UB	Unicast-Broadcast
UUB	Unicast-Unicast-Broadcast
VIM	Virtualized Infrastructure Manager
VNF	Virtual Network Function
VNFaaS	Virtual Network Function as a Service
VNFM	Manager
VNPaaS	Virtual Network Platform as a Service
WS	Web Services

Lista de Símbolos

P_A	Query Accuracy Probability
T_{MR}^{L}	Upper bound on Mistake Recurrence Time
T_{MR}	Mistake Recurrence Time
T_{D}^{U}	Upper bound on the Detection Time
T_D	Detection Time
T_{M}^{U}	Upper bound on Mistake Duration
T_M	Mistake Duration

Capítulo 1

Introdução

Redes de computadores são formadas por diversos dispositivos de rede que são implementados e provisionados em hardware e software específicos. Estes dispositivos, também denominados de *middleboxes*, são dependentes da tecnologia oferecida pelos seus fabricantes. Os *middleboxes* são desenvolvidos para prover serviços importantes como, roteamento, gerenciamento, filtragem de dados, segurança, entre outros [Xilouris et al., 2014]. Se, por um lado, *middleboxes* introduzem benefícios, por outro, são responsáveis pela dificuldade em prover novos serviços de rede devido a natureza destes dispositivos serem dedicados e apresentarem uma arquitetura proprietária [Han et al., 2015]. Os *middleboxes* também representam, para as empresas, uma importante fração de despesas operacionais (OPEX - *OPerational EXpenditures*) e despesas de capital (CAPEX - *CAPital EXpenditures*) [Cotroneo et al., 2014]. OPEX se refere a despesas como a manutenção de equipamentos, ao passo que CAPEX inclui despesas para a aquisição de novos equipamentos de rede. Além disso, *middleboxes* são pouco maleáveis, ineficientes em termos de custo de energia, difíceis de gerenciar e de solucionar problemas [Sherry et al., 2012].

Funções Virtualizadas de Rede (NFV - *Network Function Virtualization*) surgem para contornar estes problemas. Em geral, as NFVs exploram tecnologias de virtualização para transformar equipamentos de redes em entidades virtuais [ETSI, 2015b]. Estas entidades podem ser implementadas em conjunto com outras tecnologias emergentes, como a execução em redes SDN (*Software Defined Network*) [Han et al., 2015]. Neste contexto, as NFVs são utilizadas para prover soluções no plano de controle como também no plano de dados da rede. Indústria, academia e organismos de padronização têm mostrado interesse em projetos para NFV [Bondan et al., 2014], pois podem prover novos serviços aos usuários em um rápido ciclo de desenvolvimento, reduzindo custos de aquisição de equipamentos físicos e o custo global necessário para o gerenciamento de um grande número de dispositivos [Haleplidis et al., 2014, Xilouris et al., 2014].

Uma NFV implementa funções de rede através de técnicas de virtualização de software, que são executadas em hardware de prateleira. Em outras palavras, as funcionalidades dos dispositivos de rede são virtualizadas e podem ser instanciadas/executadas por demanda quando necessárias e destruídas quando estiverem ociosas. Dessa forma, uma NFV pode estabelecer novos mecanismos para operar e desenvolver funções de rede, com diversos benefícios, inclusive a redução de espaço físico e a economia de energia, se comparada aos tradicionais *middleboxes* [Ferrer Riera et al., 2014]. Por outro lado, ainda há diversos desafios a serem contornados no contexto das NFVs, entre os quais se destacam a necessidade de prover mecanismos para tolerância a falhas, em particular métodos para detecção, isolamento, recuperação e sincronização consistente das funções virtualizadas [Cotroneo et al., 2014, Canini et al., 2015].

Com base no exposto, o presente trabalho tem por objetivo tirar proveito destas novas tecnologias para implementar funções virtualizadas de rede para serem executadas dentro da própria rede. Em especial, o conjunto de contribuições proposto está focado na implementação de sistemas tolerantes a falhas para garantir as operações contínuas de aplicações distribuídas. As contribuições tratam de assuntos como detectores de falhas não confiáveis, mais precisamente da qualidade de serviço oferecida pelos detectores, tais como, ajustar parâmetros de monitoramento de acordo com as necessidades de cada aplicação, como também propor o seu funcionamento dentro da rede através de uma NFV ou próximo dela, como sua implementação em uma MIB (*Management Information Base*) SNMP (*Simple Network Management Protocol*). Ainda no contexto dos detectores, propomos uma estratégia para melhorar a eficiência para o cômputo do *timeout*, em especial melhorar o tempo de detecção e o número de falsas suspeitas. Outra contribuição explora a utilização de uma função virtualizada de rede para garantir uma sincronização consistente das informações manipuladas por múltiplos controladores SDN. Por fim, apresentamos uma função virtualizada de rede que garante a entrega confiável e ordenada das mensagens que são transmitidas por difusão pelas aplicações que executam no ambiente.

Inicialmente, neste trabalho é descrita a implementação de um serviço adaptativo para detecção de falhas. Detectores de falhas têm por objetivo encapsular as premissas temporais do sistema para detectar falhas de processos. Os detectores de falhas foram propostos por Chandra e Toueg [Chandra e Toueg, 1996] para contornar o problema da impossibilidade de resolver o consenso em sistemas assíncronos sujeitos a falhas por *crash* [Fischer et al., 1985]. Isso se deve à impossibilidade em determinar quando um processo está falho ou simplesmente mais lento que os demais. Dessa maneira, os detectores de falhas monitoram os processos efetuando hipóteses temporais sobre atrasos no sistema e disponibilizando informações sobre os estados destes processos. Chandra e Toueg demonstraram que, dependendo das propriedades garantidas pelo detector, é possível executar o consenso em sistemas distribuídos assíncronos sujeitos a falhas *crash*.

O serviço para detecção de falhas proposto é implementado como uma NFV (NFV-FD - *Failure Detector*) para obter os estados dos processos e dos dispositivos de comunicação em uma rede SDN [Turchetti e Duarte Jr., 2015b, Turchetti e Duarte Jr., 2015a, Turchetti e Duarte Jr., 2017]. Além disso, foi também implementado e é descrito um serviço para detecção de falhas para a Internet (IFDS - *Internet Failure Detection Service*), que pode ser configurado de acordo com as necessidades de QoS (*Quality of Service*) das aplicações [Turchetti e Duarte Jr., 2014, Turchetti et al., 2016a, Turchetti et al., 2016b].

A NFV-FD implementa o serviço para detecção de falhas transformando as funcionalidades do detector em uma entidade virtual que pode ser executada em uma rede SDN. Neste contexto, a NFV-FD pode utilizar os benefícios da rede SDN através do compartilhamento de informações fornecidas por um controlador da rede. O compartilhamento destas informações auxilia na obtenção dos atributos dos processos a serem monitorados, bem como permite determinar o próprio estado destes processos. Além disso, são também obtidas informações de monitoramento dos estados dos enlaces de comunicação, a partir dos próprios dispositivos de rede (*switches*).

A NFV-FD trabalha na formação de uma visão que indica quais processos estão suspeitos de terem falhado. Esta visão dos estados pode ser consultada por uma aplicação distribuída responsável por tomar decisões. Em outras palavras, a NFV-FD possibilita a execução de algoritmos distribuídos tolerantes a falhas. Como exemplo deste tipo de aplicação, pode-se citar um algoritmo para a difusão confiável de mensagens. Um algoritmo para a difusão confiável foi utilizado como estudo de caso. O trabalho ainda apresenta e detalha questões sobre a implementação da NFV. Em especial, discutimos, apresentamos resultados e implementamos a

NFV-FD de diferentes maneiras: dentro e fora do controlador SDN, utilizando máquinas virtuais tradicionais e hospedada em *containers*. O desempenho do serviço de detecção de falhas e o impacto da NFV na utilização dos recursos para seu processamento, são também avaliados neste trabalho.

Na outra proposta, denominada de IFDS, o objetivo é oferecer um serviço para detecção de falhas na Internet que pode ser configurado de acordo com as necessidades de QoS de cada aplicação. O serviço proposto é composto por uma MIB SNMP denominada *fdMIB*, através da qual são monitorados os estados dos processos utilizando troca de mensagens. A *fdMIB*, além de armazenar informações importantes que podem ser consultadas via agentes SNMP, também executa a própria atividade de monitoramento dos processos. O monitoramento pode ocorrer em redes locais e através de múltiplos sistemas autônomos via Internet. Considerando a rede local, o monitoramento dos processos ocorre por troca de mensagens via agentes SNMP, ao passo que, o monitoramento em redes locais distintas via Internet, ocorre com o auxílio de *Web Services* que assumem o papel de *gateway* SNMP.

Para garantir parâmetros de QoS definidos pelas aplicações, o IFDS implementa uma estratégia que permite que uma aplicação especifique suas necessidades, por exemplo, a aplicação pode requisitar um limite de tempo para a detecção de uma falha, e o IFDS ajusta os parâmetros de monitoramento do detector de falhas com base nestas requisições de entrada. Em outras palavras, com base nos parâmetros fornecidos e no comportamento observado da rede, o IFDS é configurado para tentar cumprir com os parâmetros solicitados pela aplicação. Um algoritmo é proposto que recebe os valores de QoS fornecidos pelas aplicações para ajustar os parâmetros de monitoramento do detector de falhas. Além disso, são propostas duas estratégias (η_{max} e η_{GCD}) que permitem compartilhar o serviço de detecção de falhas entre múltiplas aplicações. Um protótipo do serviço proposto foi implementado e resultados experimentais mostram as funcionalidades do IFDS através do compartilhamento dos estados dos processos em múltiplos sistemas autônomos. Os resultados experimentais mostram também os benefícios dos ajustes realizados pelo IFDS com base nos valores de QoS, incluindo a possibilidade de omitir para as aplicações, falsas suspeitas cometidas pelo detector de falhas.

Nos estudos realizados no âmbito dos detectores de falhas, em particular quanto ao cômputo do *timeout*, surge outra proposta onde é descrita uma estratégia para melhorar a eficiência do mecanismo de detecção de falhas, em especial o tempo de detecção e o número de falsas suspeitas cometidas pelo detector. A estratégia é denominada de $tuning_{\phi}$ e reajusta o valor do *timeout* de acordo com os tempos de comunicação calculados, de maneira a refletir o comportamento real da rede. Na prática, adaptamos o cálculo proposto por Jacobson [Jacobson, 1988] ajustando o peso da constante responsável por multiplicar o valor correspondente aos tempos de comunicação. Dessa maneira, $tuning_{\phi}$ torna o cálculo de Jacobson mais fiel ao comportamento do ambiente. Além disso, a própria estratégia é responsável por indicar pesos para valores que, no algoritmo original, são constantes.

A estratégia $tuning_{\phi}$ é avaliada no contexto de detecção de falhas de processos na Internet, cenário em que a comunicação pode ocorrer com longos períodos de atraso. Mesmo assim, é possível verificar, através dos experimentos, que $tuning_{\phi}$ reduz de forma expressiva os erros cometidos pelo serviço de detecção, não atingindo 1% do número de falsas suspeitas cometidas pelo algoritmo original. Além disso, $tuning_{\phi}$ apresenta uma redução no tempo de detecção de falhas, mantendo um bom desempenho no tempo médio para correção de falsas suspeitas.

Em uma rede SDN que utiliza um plano de controle distribuído, há a necessidade de realizar a sincronização entre os controladores SDN. Porém, a sincronização entre os múltiplos controladores distribuídos não é uma tarefa trivial. Neste trabalho é proposta uma solução

para a sincronização consistente do plano de controle em redes SDN através da implementação de uma função virtualizada de rede (VNF – *Virtual Network Function*) denominada de *VNF-Consensus*. *VNF-Consensus* implementa o algoritmo Paxos e com sua utilização os controladores ficam desacoplados e podem executar em paralelo suas atividades no plano de controle. Cada controlador SDN possui acesso a uma instância da *VNF-Consensus* através da qual poderá receber decisões e enviar dados para serem sincronizados. Dessa maneira, todas as decisões executadas pela *VNF-Consensus* são sistematicamente executadas sem a atuação direta dos controladores. A presente solução difere das demais encontradas na literatura, por não estar implementada dentro dos *switches* [Schiff et al., 2016, Dang et al., 2015] e, nem sequer, nos controladores SDN [Koponen et al., 2010, Canini et al., 2015, Ho et al., 2016].

As vantagens dessa abordagem são: (*i*) os controladores ficam desacoplados e podem executar em paralelo suas atividades no plano de controle; (*ii*) o plano de controle é sincronizado sem aumentar a carga de trabalho dos controladores, pois a *VNF-Consensus* é executada como entidade externa aos controladores, não utilizando os mesmos recursos computacionais; (*iii*) como os controladores não participam diretamente nas decisões do Paxos, o consenso consegue garantir, independentemente do número de controladores operacionais, a conclusão de seus serviços; (*iv*) por fim, a solução proposta não exige nenhuma alteração no funcionamento padrão do protocolo *OpenFlow* ou nos *switches* SDN. Resultados experimentais mostram o desempenho da *VNF-Consensus* e seu impacto na utilização dos recursos, como também os benefícios obtidos por executar o serviço sem a participação direta dos controladores SDN.

Por último, tratamos da questão da difusão confiável, que consiste em garantir a entrega das mensagens transmitidas na rede para todos os processos corretos do sistema [Défago et al., 2004]. Essa comunicação, conhecida também como *reliable broadcast*, é fundamental para a construção de aplicações distribuídas tolerantes a falhas [Pedone e Schiper, 2003]. Neste contexto, propomos *AnyBone*, uma NFV que oferece as primitivas de difusão confiável para garantir a entrega ordenada das mensagens transmitidas em uma rede SDN. Isto é, a própria rede garante a entrega confiável das mensagens e de maneira ordenada. Em especial, *AnyBone* garante a ordem das mensagens por um sequenciador localizado dentro da rede, enquanto que a entrega é realizada por uma biblioteca localizada nos processos envolvidos na comunicação.

Sendo assim, *AnyBone* divide a responsabilidade entre dois componentes que operam integrados. Um dos componentes é uma Função Virtualizada de Rede denominada de *VNF-Sequencer*. A *VNF-Sequencer* garante a entrega atômica para todos os processos exercendo o papel do sequenciador, o qual irá gerenciar todas as transmissões. O outro componente é uma biblioteca denominada de *RBCast* e oferece uma API (*Application Programming Interface*) para as aplicações trocarem mensagens utilizando as primitivas de difusão confiável. A proposta foi implementada e os resultados experimentais mostram que *AnyBone* consegue garantir a ordem atômica das mensagens pelo uso do sequenciador, essa abordagem facilita a implementação do serviço. Por outro lado, há um custo a ser pago em relação à latência para a entrega das mensagens. O custo da latência é apresentado em diferentes cenários. Por fim, foi medido o *throughput* aumentando o número de transmissores, *switches* e controladores SDN, onde o sequenciador não demonstrou ser um gargalo.

O restante deste trabalho está organizado da seguinte forma. No Capítulo 2 são apresentados conceitos sobre NFV, incluindo seus propósitos, uma iniciativa existente para a padronização de uma arquitetura para NFV e trabalhos relacionados. No Capítulo 3 são abordados conceitos que tratam da implementação de sistemas distribuídos confiáveis, em particular são introduzidos os conceitos sobre detectores de falhas não confiáveis, protocolo de consenso Paxos e algoritmo para difusão confiável (*reliable broadcast*) e suas variações. Nos capítulos 4, 5 e 6, são apresentados os serviços IFDS, NFV-FD e $tuning_{\phi}$, respectivamente. A *VNF-Consensus* é

detalhada no Capítulo 7. Por fim, *AnyBone* é descrito no Capítulo 8 e as conclusões desta tese são apresentadas no Capítulo 9.

Capítulo 2

Virtualização de Funções de Rede

O objetivo deste capítulo é descrever as funções virtualizadas de rede (*Network Function Virtualization* - NFVs) que são introduzidas na Seção 2.1. Na Seção 2.2 é apresentada uma arquitetura para NFV que foi proposta como uma iniciativa de padronização. Por fim, na Seção 2.3 são descritos diversos trabalhos que visam o desenvolvimento de soluções em NFV com diferentes propósitos e cenários.

2.1 *Network Function Virtualization*: Introdução

Redes de computadores de grande porte são formadas por um grande número de dispositivos especializados denominados *middleboxes* [Sherry et al., 2012]. Como exemplo, pode-se citar *firewalls*, detectores de intrusão, balanceadores de carga, controladores e aceleradores de fluxo, entre diversos outros. Estes dispositivos possuem suas funcionalidades implementadas em hardware e software específicos. *Middleboxes* oferecem benefícios importantes, tais como, ferramentas de segurança, otimização do desempenho, garantia de qualidade de serviço, entre outros. Entretanto, estes dispositivos tornam a infraestrutura da rede complexa e com alto custo de gerenciamento. Os autores em [Sherry et al., 2012] avaliam e demonstram a complexidade de se gerenciar diversos *middleboxes*, considerando o tamanho da rede, diversidade de fabricantes e necessidade de manutenções periódicas, e concluem que uma infraestrutura composta por *middleboxes* possui um custo elevado para a manutenção, atualização e é difícil de gerenciar e de tratar falhas.

Em síntese, níveis de investimento consideráveis são necessários para desenvolver, gerenciar e manter dispositivos de rede baseados em hardware. De fato, eles constituem uma importante fração de despesas operacionais (*OPerational EXpenditures - OPEX*) e despesas de capital (*CAPital EXpenditures - CAPEX*) das redes de computadores [Cotroneo et al., 2014]. Além disso, *middleboxes* possuem limite de flexibilidade, são ineficientes em termos de consumo de energia e problemas, em geral, não são fáceis de serem resolvidos [Sherry et al., 2012].

Funções virtualizadas de rede surgem como uma alternativa para a implementação de *middleboxes*. Em geral, as NFVs exploram tecnologias de virtualização para transformar equipamentos de redes em entidades virtuais [ETSI, 2015b]. A ideia é fornecer funções de redes implementadas exclusivamente em software. Indústria, academia e organismos de padronização têm mostrado interesse em projetos para NFV [Bondan et al., 2014], pois é possível estabelecer novos mecanismos para operar e desenvolver serviços de rede, de maneira mais simplificada, econômica e eficiente [Ferrer Riera et al., 2014].

Essencialmente, uma NFV implementa funções de rede através de técnicas de virtualização de software, que são executadas em hardware de prateleira. Este cenário pode ser

visto na Figura 2.1, onde dispositivos de rede são migrados para uma solução baseada em NFV utilizando tecnologias de virtualização. Na Figura 2.1 são ilustradas funções de rede como, roteador, dispositivo NAT (*Network Address Translation*), filtro de pacotes, entre outros, que são migradas para um ambiente em NFV onde as funções virtualizadas podem ser instanciadas/executadas por demanda quando necessárias e destruídas quando estiverem ociosas. Dessa forma, uma NFV pode estabelecer novos mecanismos para operar e desenvolver funções de rede, com diversos benefícios, inclusive a redução de espaço físico e a economia de energia [Ferrer Riera et al., 2014].

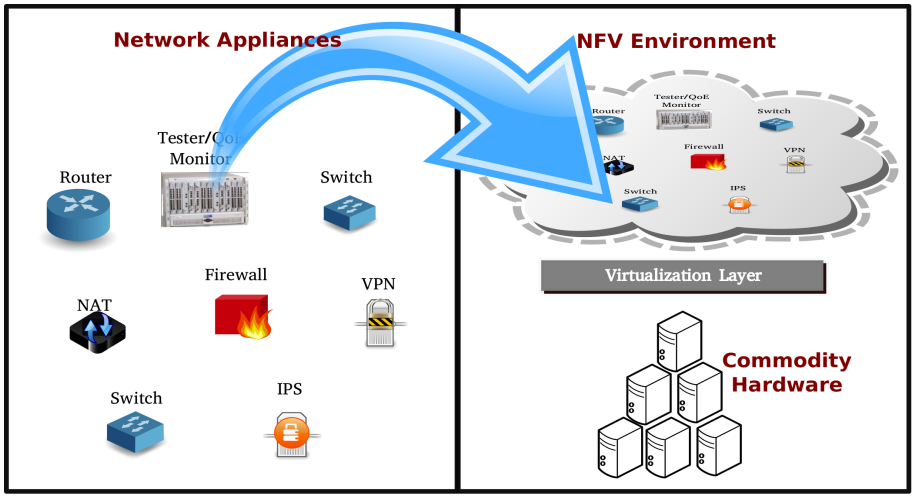


Figura 2.1: De dispositivos de rede baseados em hardware para soluções baseadas em NFV [Han et al., 2015].

O ambiente para a execução das NFVs deve ser eficiente em diversos aspectos, tais como, possibilitar a integração de soluções pertencentes a diferentes desenvolvedores, como também a integração entre funções virtualizadas e não virtualizadas, prover interfaces para o gerenciamento e orquestração das diversas NFVs, oferecer suporte para serviços dinâmicos, reconfiguráveis, garantir a escalabilidade, entre outros.

Na próxima seção é apresentada uma arquitetura para NFV proposta pela ETSI (*European Telecommunications Standards Institute*) que endereça diversas questões incluindo o desenvolvimento, execução, orquestração e gerenciamento no contexto de NFVs.

2.2 Uma Arquitetura para NFV Proposta Pela ETSI ISG

Uma arquitetura para NFV deve prover mecanismos para executar, gerenciar e orquestrar múltiplas NFVs em uma infraestrutura padronizada e aberta, além disso, as funções virtualizadas de rede devem executar em um ambiente seguro e que ofereça disponibilidade, de maneira que provedores possam oferecer serviços que satisfaçam as necessidades de seus clientes [ETSI, 2015c].

Neste sentido, o grupo ISG (*Industry Specification Group*) criado pela ETSI, tem apresentado esforços para a padronização de uma arquitetura em alto nível para o desenvolvimento de NFV. O objetivo é orientar partes interessadas da indústria, incluindo operadores de rede, fornecedores de soluções em telecomunicação e provedores de serviços, para a utilização de uma arquitetura padronizada e consistente. O documento apresentado pela [ETSI, 2015b] descreve como os elementos necessários para compor uma NFV podem ser implementados, considerando aspectos de interoperabilidade e de padronização. A Figura 2.2 ilustra o *framework* NFV

em alto nível, onde pode-se destacar quatro blocos funcionais: *Orchestrator*, *VNF Manager*, *Virtualization Layer* e *Virtualization Infrastructure Manager*, descritos a seguir.

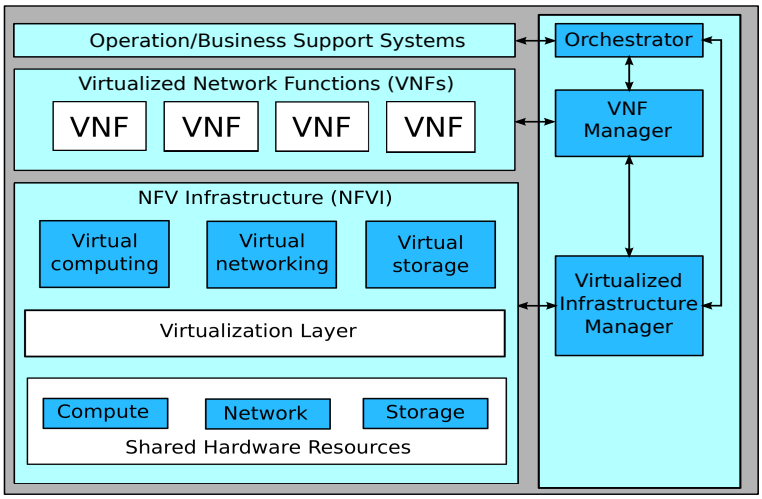


Figura 2.2: Arquitetura NFV especificada pela ETSI ISG.

O *Orchestrator* ilustrado na Figura 2.2 é responsável pelo gerenciamento e orquestração dos recursos de software e da infraestrutura de hardware virtualizada, objetivando controlar os serviços de rede presentes na NFVI (*NFV Infrastructure*). O *VNF Manager* é responsável por controlar todo o ciclo de vida das VNFs[1] (*Virtual Network Functions*). Por exemplo, ele é responsável por instanciar, escalonar e terminar/interromper a execução de uma VNF. A *Virtualization Layer* abstrai os recursos físicos e abriga as VNFs para a infraestrutura virtualizada, em particular, ela assegura que o ciclo de vida da VNF seja independente da plataforma de hardware subjacente. Este tipo de funcionalidade é tipicamente oferecida na forma de máquinas virtuais e seus monitores (hipervisor).

Por fim, o *Virtualized Infrastructure Manager* é usado para virtualizar, gerenciar a configuração de recursos (computadores, recursos de armazenamento, entre outros) e controlar as interações com as VNFs, em específico, alocar as máquinas virtuais nos hipervisores, gerenciar a conectividade da rede e analisar o desempenho coletando informações sobre falhas na infraestrutura.

De fato, a especificação da arquitetura proposta pela ETSI traz benefícios para o desenvolvimento de novas soluções em NFV, evidência disso são as diversas provas de conceitos apresentadas[2] e baseadas na arquitetura da Figura 2.2. Além disso, esta arquitetura é considerada como ponto de partida para a criação do projeto OPNFV (*Open Platform for NFV*), que tem por objetivo o desenvolvimento colaborativo de uma plataforma de referência *open source* para estimular inovações em NFV. O projeto OPNFV foi criado com estreita colaboração do grupo ISG, sendo também apoiado pela *Linux Foundation* em conjunto com um consórcio de empresas [OPNFV, 2015].

Outra importante contribuição proposta pela ETSI [ETSI, 2015d] é a especificação de um *framework* para provisionar o gerenciamento e orquestração de funções virtualizadas de rede, objetivando prover estas atividades no nível operacional. Além disso, segundo o documento, a especificação do *framework*, denominado de NFV-MANO (*Network Functions Virtualization*

[1]Uma VNF é a implementação em software de uma função de rede que pode ser empregada em uma infraestrutura NFV.

[2]http://www.etsi.org/technologies-clusters/technologies/nfv/nfv-poc

Management and Orchestration), deve servir como ferramenta para orientar e promover novos trabalhos no contexto das NFVs.

O NFV-MANO é apresentado através de uma arquitetura que oferece funções que permitem gerenciar todo o ciclo de vida das VNFs, isto é, criar/instanciar uma VNF, aumentar ou reduzir sua capacidade de execução e utilização de recursos, permitir a atualização e configuração da VNF e, por fim, terminar/interromper a execução da VNF, liberando todos os recursos associados.

O NFV-MANO é formado por grupos de blocos funcionais que foram descritos na arquitetura anterior (ver Figura 2.2), mas que agora são detalhados na visão da orquestração e do gerenciamento para NFVs. Como apresentado na Figura 2.3, são especificados 3 blocos funcionais: *Virtualized Infrastructure Manager* (VIM), *NFV Orchestrator* (NFVO) e VNF *Manager* (VNFM). Estes blocos são descritos na sequência.

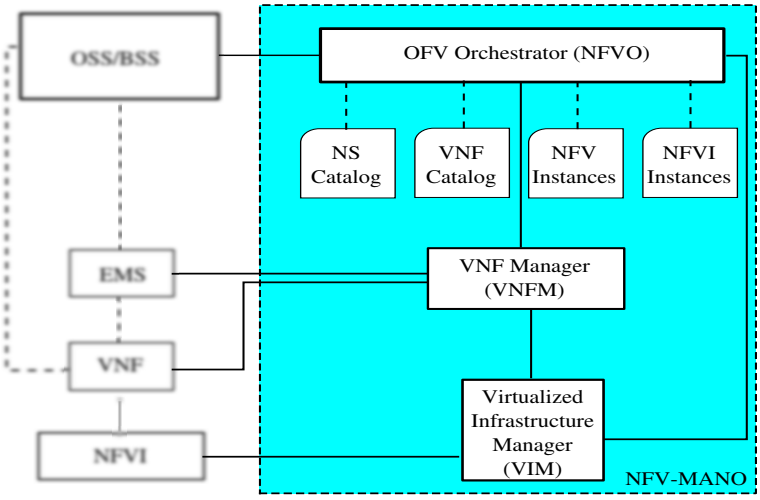


Figura 2.3: NFV-MANO: uma arquitetura para orquestração e gerenciamento de NFV.

O bloco NFVO possui duas principais responsabilidades: a orquestração dos recursos da NFVI e o gerenciamento do ciclo de vida dos serviços de rede (NS - *Network Services*). O bloco NFVO deve coordenar as múltiplas instâncias de VNFs, em particular permitir conexões entre as VNFs e gerenciar mudanças dinâmicas de configuração, por exemplo, escalar a capacidade dos serviços de rede quando necessário. Estão previstas também interfaces que podem ser usadas por entidades externas, tal como OSS (*Operations Support System*) e BSS (*Business Support System*), para permitir a integração com a arquitetura NFV-MANO, além disso, para este mesmo propósito o NFVO deve ser capaz de utilizar as interfaces oferecidas pelas próprias entidades externas.

O bloco VNFM é responsável pelo gerenciamento da própria VNF. Neste bloco são realizadas configurações genéricas aplicáveis a qualquer tipo de VNF. Configurações iniciais como fornecer endereços de rede, notificar eventos de falhas, verificar a integridade da VNF, entre outras funcionalidades, são responsabilidades do VNF *Manager*. Além disso, a VNF *Manager* se comunica também com o EMS (*Element Management System*) responsável por manter informações para o gerenciamento de cada VNF.

VIM é responsável por controlar e gerenciar recursos da NFVI, tais como, o processamento, armazenamento e recursos de rede, usualmente pertencentes ao domínio da infraestrutura do operador, ou seja, gerenciar os recursos de software (por exemplo, manter informações de falhas nos hipervisores e máquinas virtuais) e recursos de hardware (por exemplo, informações sobre a computação, armazenamento e dispositivos de rede).

Diante do exposto nesta seção, pode-se observar que há iniciativas para soluções de problemas em NFV, em particular quando se refere a questões de padronização. Entretanto, no documento proposto pela ETSI [ETSI, 2015c] são apresentados vários desafios que ainda precisam ser superados. Neste sentido, na próxima seção são descritos trabalhos que estão relacionados com o desenvolvimento de soluções em NFV, com o objetivo de resolver problemas em diferentes contextos.

2.3 Trabalhos Relacionados

A possibilidade de desenvolver serviços através da implementação e virtualização de funções de rede, tem feito crescer a pesquisa e surgir novos projetos em NFV. Há várias soluções para a implementação de uma NFV disponíveis na literatura, que consideram sua execução em uma rede OpenFlow [Batalle et al., 2013, Fukushima et al., 2014, Vilalta et al., 2015]. A propósito, OpenFlow é o protocolo mais comum para a construção de redes SDN, possibilitando a separação entre o plano de controle e o plano de dados, definindo regras para a comunicação entre os dispositivos da rede, através de um controlador [Kreutz et al., 2015]. Uma visão geral sobre OpenFlow pode ser encontrada, por exemplo, no trabalho apresentado por Lara [Lara et al., 2014].

Outro cenário utilizado com frequência para a implementação e execução de NFVs é através do uso de plataformas de computação em nuvem. Por exemplo, CloudNFV[3] é uma plataforma aberta para a implementação de NFVs baseada em computação em nuvem e composta por um ambiente "*multi-vendor*". A plataforma integra diferentes soluções como: computação em nuvem, SDN, NFV e TMF (*Telemanagement Forum*[4] – para tratar questões de gerenciamento). Nesta abordagem, operadores podem simplificar suas redes removendo a complexidade da topologia e acelerando o processo de criação para a entrega de novos serviços. A principal ideia do CloudNFV é a virtualização de tudo (*"everything is virtual"*), isto é, virtualizar serviços, funções e recursos da nuvem, com o objetivo de reduzir custos e aumentar receitas.

Os autores em [Pham et al., 2015] utilizam a arquitetura CloudNFV para maximizar lucros e a utilização dos recursos ativos. A ideia é utilizar máquinas virtuais dinâmicas para melhorar a utilização de recursos físicos e reduzir o consumo de energia. A estratégia trabalha com a realocação de VMs (*Virtual Machines*), de acordo com a demanda de recursos em tempo real e através da inserção de recursos ociosos em modo de suspensão. Para executar esta tarefa, inicialmente são coletadas informações das aplicações e das condições dos recursos disponíveis. As informações são coletadas por uma NFVO responsável por calcular o melhor local para hospedar uma VM. Em outras palavras, a NFVO realiza migração em tempo real com o objetivo de rearranjar as VMs em locais que possam prover melhor utilização dos recursos (servidores), mantendo a consistência dos serviços.

No documento apresentado pela ETSI [ETSI, 2015a], os autores introduzem diferentes modelos para a aplicação dos conceitos de NFV em serviços tradicionais oferecidos por computação em nuvem, resultando nos seguintes modelos: *Network Functions Virtualization Infrastructure as a Service* (NFVIaaS); *Virtual Network Function as a Service* (VNFaaS); e *Virtual Network Platform as a Service* (VNPaaS). Os diferentes modelos de aplicação são apresentados por estudos de casos que possuem como objetivo esclarecer regras e interações para a entrega de serviços via NFVs usando computação em nuvem. Os detalhes dos modelos de serviços apresentados são destinados a vários tipos de entidades, tais como, provedores de

[3]https://www.cloudnfv.com/

[4]https://www.tmforum.org/

serviços, empresas de telecomunicação, consumidores e pesquisadores. Em síntese, o objetivo do documento é introduzir modelos de serviços como desafios para novos estudos de casos através do desenvolvimento de NFVs no contexto dos modelos citados.

Vilalta [Vilalta et al., 2015] utilizam NFVs através de uma arquitetura em nuvem para disponibilizar serviços executados por um PCE (*Path Computation Element*). PCE é um elemento responsável pelo mecanismo de cálculo de rota para criação de caminhos em uma rede de transporte. O objetivo dos autores é escalar o serviço oferecido por estes elementos. Para isso, não se utiliza servidor PCE dedicados e suas funcionalidades são virtualizadas (vPCE) e transportadas para um sistema em nuvem (OpenStack). Os resultados da implementação do PCE como uma NFV, demonstraram que o vPCE foi capaz de garantir um tempo médio de processamento, mesmo perante a picos de solicitações para o cômputo dos caminhos.

Cotroneo [Cotroneo et al., 2014] discutem os desafios para se obter confiabilidade no contexto de uma infraestrutura para a execução de NFVs. O trabalho descreve ferramentas para a injeção de falhas, bem como diretrizes sistemáticas para avaliar elementos relacionados a uma NFVI. No estudo, os autores relatam que potenciais causas de falhas em uma função virtualizada de rede, proveem de diversos fatores como: falhas de hardware, falhas de software e falhas operacionais. A proposta considera, hipoteticamente, a execução dos testes em um ambiente em nuvem, onde os seguintes componentes são mapeados para avaliação: máquinas virtuais, gerenciador da nuvem e hipervisor. Para cada elemento, um conjunto de ferramentas para a injeção de falhas é proposto. Vale ressaltar que este conjunto de ferramentas é somente descrito no trabalho, não sendo apresentado nenhum experimento específico. Neste sentido, acredita-se que o trabalho carece de uma avaliação prática, tanto considerando as ferramentas propostas quanto as diretrizes sistemáticas relatadas no trabalho.

Batalle [Batalle et al., 2013] propõem uma função virtualizada para o encaminhamento de pacotes. A NFV separa os fluxos dos pacotes do protocolo IPv4 (*Internet Protocol version* 4) e IPv6 (IP *version* 6), oferecendo roteamento entre domínios através de redes OpenFlow. Resumidamente, a ideia é criar um módulo dentro do controlador responsável por selecionar os pacotes que devem ser processados pela NFV. Os autores conseguem reduzir o fluxo de entrada nos *switches* transferindo funcionalidades (seleção dos fluxos) para uma máquina externa que implementa a função virtualizada. Segundo os autores, esta foi a primeira validação de implementação de uma NFV para auxiliar no encaminhamento de pacotes, transportando as funcionalidades para uma entidade externa aos *switches*.

Preocupados com o gerenciamento em ambientes com diversas NFVs, Fukushima et. al [Fukushima et al., 2014] propõem um *framework* que permite a um operador de rede descrever as NFs (*Network Functions*) virtualizadas através de suas características e atributos. O *framework* captura estas informações, cria relações entre as variáveis de diversas NFs e a consistência é sistematicamente validada. Como exemplo, considere duas NFs: uma para NAT e outra para prevenção de intrusão IPS (*Intrusion Prevention System*). Os pacotes que passam pela função NAT dependerão do endereço da origem (atributo) para serem processados, ao passo que a função de IPS deverá processar todos os pacotes. Considerando a existência de uma DMZ (*DeMilitarized Zone*), os pacotes originados desta rede não devem ser processados pela função NAT , ao passo que os pacotes da rede privada devem ter os endereços traduzidos. Através do *framework* a ideia é criar estas regras para que as funções virtualizadas sejam configuradas. A proposta pode ser considerada relevante, mas ainda há problemas a serem resolvidos, como exemplo, a proposta não suporta configurações dinâmicas, e o *script* gerado pelo *framework* é executado no controlador OpenFlow, o que poderá sobrecarregar as tarefas deste controlador.

Nos serviços oferecidos por nuvens computacionais, se os contratos em nível de serviço (SLA - *Service Level Agreements*) não puderem ser garantidos, por exemplo, em casos em que

a QoS não puder ser ativada, os provedores destes serviços devem sofrer penalidades. Para evitar esta situação, a infraestrutura da nuvem deve operar minimizando o desperdício dos recursos e de energia, sendo necessária a adaptação e o monitoramento constante dos serviços. Neste sentido, Buyya [Buyya et al., 2014] apresentam uma nova abordagem denominada de *Software-Defined Cloud* (SDC). SDC habilita a configuração e adaptação de recursos físicos e virtualizados da nuvem computacional, para melhor atender a demanda dos serviços. Em particular, SDC identifica e habilita as necessidades como garantir os requisitos formalizados pelos contratos de nível de serviço, como exemplo, garantir a escalabilidade, desempenho, tempo de atividade, segurança, utilização dos recursos físicos e virtualizados, entre outros.

Na Figura 2.4, pode-se observar um SDC formado pelas seguintes camadas: (1) uma rede SDN, (2) um sistema de computação em nuvem e (3) um ambiente de suporte para NFV. A camada referente à rede SDN, tem por objetivo permitir a separação do plano de dados (*data plane*) e plano de controle (*control plane*), tornando *switches* dispositivos simples para encaminhamento de pacotes, uma vez que o controle lógico é implementado em um elemento denominado de *controller*. No topo das camadas SDC observa-se o suporte para a NFV, esta camada fornece uma infraestrutura adequada para a execução de VNFs hospedadas em máquinas que são virtualizadas por um hipervisor. Por fim, para controlar estas duas camadas (SDN e NFV), o SDC é formado por uma camada intermediária onde se tem soluções de computação em nuvem, como exemplo, OpenStack[5], OpenNebula[6] e Eucaliptus[7]. Além disso, é necessário que o SDC possua mecanismos eficientes para o gerenciamento dos recursos físicos e virtuais, incluindo funções para a criação, destruição e migração destes recursos.

Com o crescente interesse pelo emprego de NFVs, tanto pela indústria quanto pela academia, outros desafios surgem como, por exemplo, questões que tratam da segurança e integridade da NFV. A ETSI criou um grupo de trabalho focado em assuntos de segurança e privacidade em NFV [ETSI, 2015e]. Os autores em [Firoozjaei et al., 2017] relatam várias ameaças, discutem diversos desafios que devem ser enfrentados pelas NFVs e sugerem algumas práticas para prover uma plataforma de execução de NFVs menos vulnerável. Usando uma outra abordagem, os autores em [Rashidi et al., 2017] propõem uma rede colaborativa baseada em NFV para defesa contra ataques de DDoS (*Distributed Denial of Service*). Neste ambiente colaborativo as redes podem redirecionar os ataques, que são caracterizados por fluxos excessivos, para serem filtrados via NFVs em outros domínios. A solução demonstra que a implementação de uma defesa colaborativa, contra esse tipo de vulnerabilidade, reduz o impacto do ataque. Os autores em [Bondan et al., 2017] propõem um *framework* para o desenvolvimento de técnicas que permitem detectar anomalias, permitindo manter a integridade das SFCs[8] (*Service Function Chains* – funções de serviços encadeadas). Um módulo proposto permite analisar elementos da NFV, bem como, sugerir ações no sentido de manter a integridade da rede. A solução interage com o NFVO (*NFV Orchestrator*) para fornecer operações reativas de segurança, como parar a VNF que apresenta anomalias. Os resultados obtidos através da implementação de uma prova de conceitos mostraram que a solução foi capaz de detectar anomalias e imediatamente sugerir ações para o orquestrador da NFV.

[5]https://www.openstack.org/

[6]http://opennebula.org/

[7]https://www.eucalyptus.com/

[8]Responsável por conectar as funções em uma ligação ordenada dos serviços.

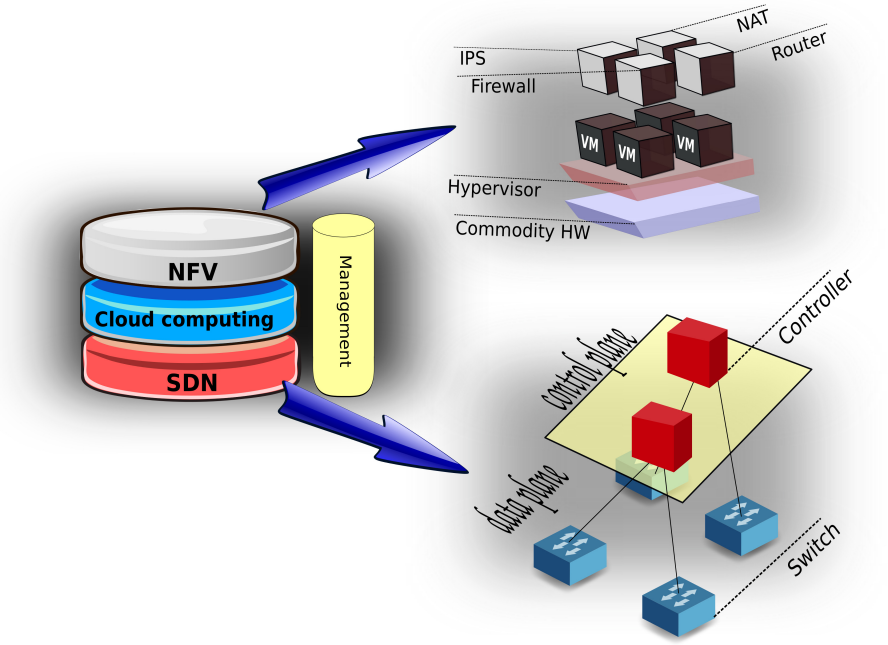


Figura 2.4: Software-Defined Clouds formado por três camadas.

2.4 Considerações Parciais

Neste capítulo foram introduzidos conceitos sobre NFV, em particular descrevendo seu propósito de solucionar problemas através da utilização de tecnologias de virtualização para fornecer funções de redes exclusivamente em software. Como iniciativa para a padronização de uma arquitetura para NFV, o grupo ISG da ETSI propôs uma arquitetura com o objetivo de prover mecanismos para executar, gerenciar e orquestrar múltiplas NFVs em uma infraestrutura padronizada e aberta. Por fim, foram apresentados trabalhos em NFV onde são propostas melhorias para a entrega de serviços auxiliando vários tipos de entidades, tais como, provedores de serviços, empresas de telecomunicação, consumidores e pesquisadores.

Capítulo 3

Sistemas Distribuídos Confiáveis

Este capítulo trata de soluções para resolver problemas fundamentais em sistemas distribuídos tolerantes a falhas, sendo eles utilizados para a construção real de aplicações que desejam garantir suas operações mesmo na presença de falhas. São descritos alguns algoritmos utilizados como blocos básicos para a construção de aplicações confiáveis. Em especial, o objetivo deste capítulo é descrever os conceitos básicos sobre algoritmos que são desenvolvidos nos capítulos subsequentes desta tese.

Sendo assim, na Seção 3.1 são introduzidos os detectores de falhas não confiáveis, em particular são apresentados os propósitos, as propriedades, os modelos utilizados para monitoramento dos processos (Seção 3.1.1) e as métricas que permitem avaliar a qualidade do serviço de detecção (Seção 3.1.2).

Na sequência, Seção 3.2, são introduzidas as características de funcionamento do protocolo de consenso, tendo como foco o algoritmo Paxos [Lamport, 1998]. A grosso modo, um protocolo de consenso permite que um conjunto de processos concordem sobre um valor único com base em valores propostos inicialmente, considerando que esses valores iniciais podem ser diferentes para cada processo. O algoritmo Paxos pode ser utilizado para garantir a decisão única entre estes processos.

Por fim, a Seção 3.3 aborda um caso especial de comunicação que ocorre quando um processo deseja transmitir uma mensagem que precisa ser entregue, garantidamente, a todos os demais processos corretos do sistema. Essa comunicação é conhecida como difusão confiável (*reliable broadcast*). Além disso, são apresentadas outras variações da difusão confiável, como aquelas que descrevem propriedades que garantem além da entrega, também a ordem das mensagens para as aplicações.

3.1 Detectores de Falhas Não Confiáveis

A noção de detectores de falhas foi proposta por Chandra e Toueg [Chandra e Toueg, 1996] e sua abstração possibilita encapsular o indeterminismo temporal dos sistemas assíncronos [Fischer et al., 1985]. O indeterminismo, também conhecido como impossibilidade FLP (das iniciais de seus autores) [Fischer et al., 1985], suscita que, devido à falta de conhecimento temporal dos sistemas assíncronos, no caso da falha de algum processo, não há algoritmos determinísticos que possam garantir a resolução do algoritmo de consenso. Isso se deve à impossibilidade em determinar quando um processo está falho ou simplesmente mais lento que os demais.

Um detector de falhas permite encapsular o indeterminismo por indicar os estados de cada processo participante do sistema. Os estados indicam se um processo está incorreto ou

não falho de acordo com o monitoramento que ocorre por troca de mensagens e respeitando limites de tempo (*timeout*). Um detector de falhas pode ser acessado por um processo p_i para obter informações referentes aos estados de outros processos. A informação retornada por um detector de falhas pode ser incorreta, isto é, um processo não falho pode ser considerado suspeito e vice-versa. Além disso, um detector de falhas pode fornecer informações inconsistentes divergindo de outros detectores, por exemplo, em um dado momento t, é possível que um FD_i[1] considere um processo p_x como suspeito, ao passo que, outro detector de falhas FD_j considere o mesmo processo p_x como não suspeito. Devido a situações como estas, detectores de falhas foram definidos como não confiáveis.

Para determinar as características dos detectores de falhas, Chandra e Toueg os classificam através de duas propriedades: (1) completude (*completeness*) e (2) precisão (*accuracy*). A completude caracteriza a capacidade do detector de falhas suspeitar de todos os processos falhos (*crashed*), enquanto que a precisão caracteriza a capacidade do detector de falhas não suspeitar de processos não falhos (corretos). A caracterização destas propriedades definidas por Chandra e Toueg [Chandra e Toueg, 1996] prevê dois tipos de completude e quatro tipos de precisão, sendo descritas a seguir:

- (1) Strong Completeness: em algum instante futuro, *todo* processo falho será permanentemente suspeito por *todos* processos corretos.
- (1) Weak Completeness: em algum instante futuro, *todo* processo falho será permanentemente suspeito por *algum* processo correto.
- (2) Strong Accuracy: *nenhum* processo é suspeito antes de ter falhado.
- (2) Weak Accuracy: existe *pelo menos um* processo correto que jamais será suspeito.
- (2) Eventual Strong Accuracy: em algum instante futuro, *todos* os processos são considerados suspeitos somente após falharem.
- (2) Eventual Weak Accuracy: em algum instante futuro, *algum* processo correto nunca é suspeito antes de ter falhado.

Considerando a completude e precisão de maneira isolada, é trivial garantir estas propriedades mesmo na sua condição mais forte. Por exemplo, para a completude, basta que o detector suspeite permanentemente de todos os processos, ao passo que, para a precisão, basta que o detector nunca suspeite de qualquer processo. Entretanto, na prática, um detector de falhas precisa combinar as duas propriedades para satisfazer garantias tanto de completude quanto de precisão. Desta maneira, realizando uma combinação entre as propriedades (1) e (2), podem ser deduzidas oito classes de detectores de falhas que são apresentadas na tabela 3.1.

Cada classe pode ser identificada de acordo com suas características. A seguir são comentadas algumas das principais classes. Um detector $\mathcal{P}$ chamado Perfeito deve satisfazer as propriedades de Strong Completeness e Strong Accuracy. Já o detector $\diamond\mathcal{W}$ (dito "diamante W"), que contempla as propriedades de Eventual Weak Accuracy e Weak Completeness, refere-se à classe mais fraca para resolver o consenso em um sistema assíncrono com a maioria dos processos corretos. Entretanto, Strong Completeness e Weak Completeness são equivalentes, ou seja, um detector que satisfaça a propriedade de Weak Completeness pode ser transformado para um detector que satisfaça Strong Completeness [Chandra e Toueg, 1996]. Esta equivalência pode ser implementada através da propagação de uma suspeita para todos os outros detectores,

[1]Detector de falhas do processo p_i.

Tabela 3.1: Relação das classes de detectores de falhas.

	Accuracy (2)			
Completeness (1)	Strong	Weak	Eventual Strong	Eventual Weak
Strong	Perfect $\mathcal{P}$	Strong $\mathcal{S}$	Eventually Perfect $\Diamond\mathcal{P}$	Eventually Strong $\Diamond\mathcal{S}$
Weak	$\mathcal{L}$	Weak $\mathcal{W}$	$\Diamond\mathcal{L}$	Eventually Weak $\Diamond\mathcal{W}$

fazendo com que todos suspeitem de um processo falho. Por esta razão, é dito que $\Diamond\mathcal{S}$ é a classe mais fraca dos detectores de falhas para resolver o consenso.

Fetzer [Fetzer, 2001] mostrou como um detector de falhas da classe $\Diamond\mathcal{S}$ pode ser transformado em um detector que nunca comete erros (por exemplo, $\mathcal{P}$). Esta habilidade foi garantida forçando o processo monitorado a falhar a cada suspeita do detector $\Diamond\mathcal{S}$. O protocolo proposto faz uso de *watchdogs* (cães de guarda) que atuam como injetores de falhas, garantindo que o detector de falhas jamais erre nos anúncios de falhas.

3.1.1 Implementação de Detectores de Falhas

O esquema para o monitoramento dos processos realizado por um detector de falhas é baseado em troca de mensagens. Para realizar esta tarefa, inicialmente dois algoritmos básicos foram projetados, ambos utilizando um parâmetro para periodicidade de envio de mensagens (η) e outro parâmetro para controlar limites de tempo de espera, conhecido como *timeout* (τ). Estes algoritmos foram denominados de *Push* e *Pull* [Felber et al., 1999] e são descritos a seguir.

Detector *Push*

No algoritmo de detecção *Push*, as mensagens de controle geradas pelos detectores seguem o mesmo sentido do fluxo das informações. Isto é, os processos monitorados por um detector de falhas, enviam periodicamente mensagens de *heartbeat* indicando que eles estão corretos. Caso o detector de falhas não receba uma mensagem dentro de um limite de tempo especificado, o respectivo processo monitorado passa a ser considerado falho.

A figura 3.1 apresenta a troca de informações entre um processo monitor p_j e um processo monitorado p_i. A cada mensagem `IAmAlive` recebida por p_j, ele reinicia o *timeout* correspondente ao processo emissor. Dessa forma, existem certas restrições a serem efetuadas na definição dos parâmetros η e τ. Considerando a figura 3.1, pode-se observar que τ deve ser maior que η [Sergent et al., 2001], caso contrário as mensagens *heartbeat* não chegarão ao seu destino em tempo hábil. Além disso, o detector *Push* destaca-se pelo fato de ser eficiente no número de mensagens trocadas, uma vez que o fluxo é unidirecional.

Detector *Pull*

No algoritmo de detecção *Pull*, as mensagens de controle seguem no sentido oposto ao fluxo de controle. Os processos monitorados periodicamente são questionados pelo detector de falhas com uma mensagem de *liveness request* (requisição de vida). Se um processo monitorado responder às requisições feitas pelo detector, dentro de um determinado tempo *(timeout)*, significa que ele está correto. Entretanto, se nenhuma resposta for recebida ou, se a mensagem recebida

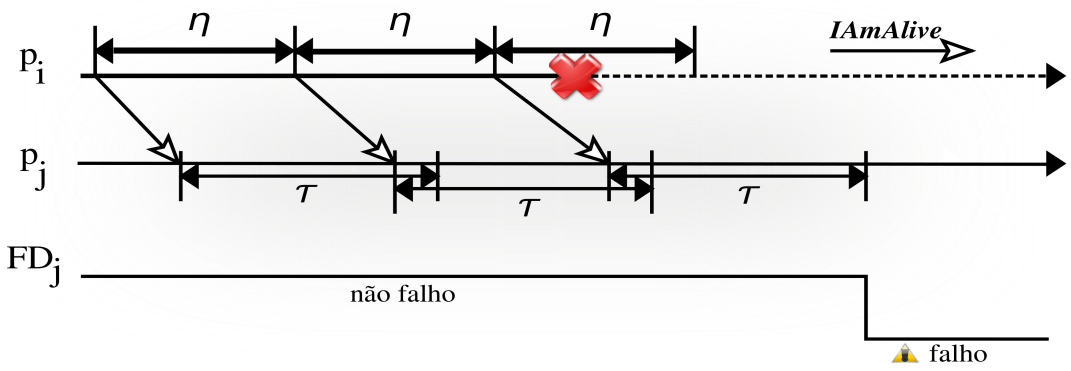


Figura 3.1: Detector de falhas *Push*.

não condizer com a respectiva requisição esperada, o processo monitorado será considerado falho.

A figura 3.2 apresenta a troca de informações entre um processo monitor p_j e um processo monitorado p_i. A cada mensagem `AreYouAlive?` recebida por p_i, ele responde ao processo monitor com uma resposta positivo `Yes`, p_j ao receber a resposta reinicia o *timeout* correspondente ao processo emissor atualizando seu estado para não falho.

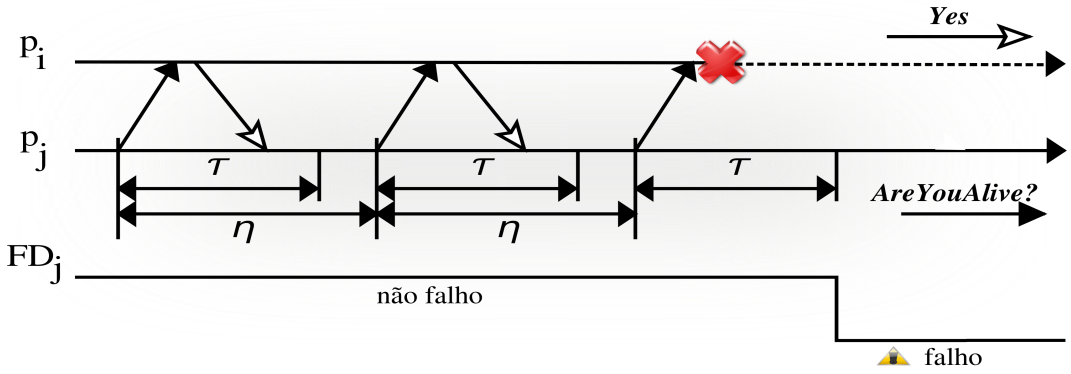


Figura 3.2: Detector de falhas *Pull*.

O algoritmo *Pull* também exige certas restrições a serem efetuadas na definição de seus parâmetros, ou seja, como a latência de detecção deste algoritmo envolve duas fases, onde a primeira corresponde ao envio de mensagens de requisição e a segunda corresponde ao recebimento das mensagens requisitadas na primeira fase, isto indica que o tempo mínimo para a configuração do τ, deverá ser maior que o tempo do seu respectivo *rtt* (*round trip time*), caso contrário, as mensagens de resposta jamais chegarão ao seu destino em tempo hábil.

3.1.2 Métricas para Detectores de Falhas

No âmbito de detectores de falhas, Chen, Toueg e Aguilera [Chen et al., 2000] propuseram um grupo de métricas que permitem avaliar a qualidade de serviço dos algoritmos de detecção de falhas. As métricas são referentes à velocidade com que os serviços conseguem suspeitar de processos falhos e à exatidão destas suspeitas. Note que a velocidade permite avaliar quão rápida uma falha pode ser detectada enquanto a exatidão avalia quão bem o detector não comete enganos.

Para especificar a QoS de detectores de falhas, as seguintes métricas primárias são definidas e ilustradas na Figura 3.3:

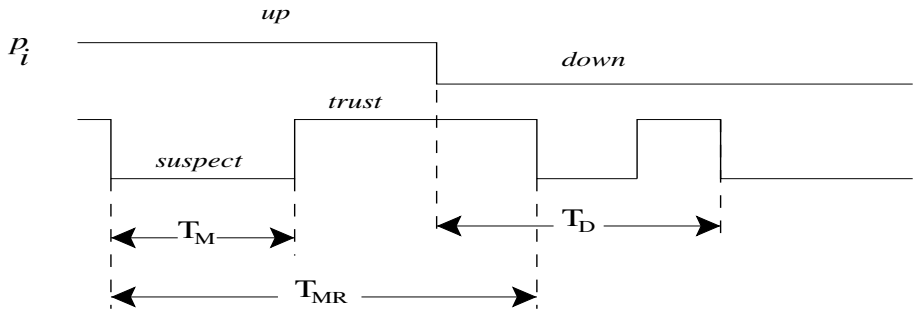


Figura 3.3: Métricas primárias.

- **Tempo de detecção (*Detection Time* - T_D).** Mede o tempo decorrido desde o instante em que p falha até o instante em que o detector de falhas suspeita permanentemente de p_i.
- **Tempo de recorrência ao erro (*Mistake Recurrence Time* - T_{MR}).** Determina o tempo entre dois erros consecutivos cometidos pelo detector. O T_{MR} é variado e inicia no instante em que ocorrer a primeira suspeita errada até a seguinte.
- **Duração de um erro (*Mistake Duration* - T_M).** Representa o tempo que um detector de falhas leva para corrigir uma suspeita incorreta.

Outras métricas são derivadas das métricas primárias, uma delas é a probabilidade de uma resposta exata (*Query Accuracy Probability* - P_A). P_A é a probabilidade com que detectores de falhas geram saídas corretas em instantes aleatórios. Observe a Figura 3.4 onde uma aplicação realiza três consultas ao FD_j para saber o estado do processo p_i, observe também que na primeira consulta o estado retornado pelo FD_j não corresponde ao estado real do processo p_i, ou seja, o detector de falhas comete um engano. Sendo assim, o valor de P_A permite que uma aplicação possa avaliar a precisão das respostas realizadas pelo detector de falhas, estas respostas se referem aos estados de cada processo monitorado.

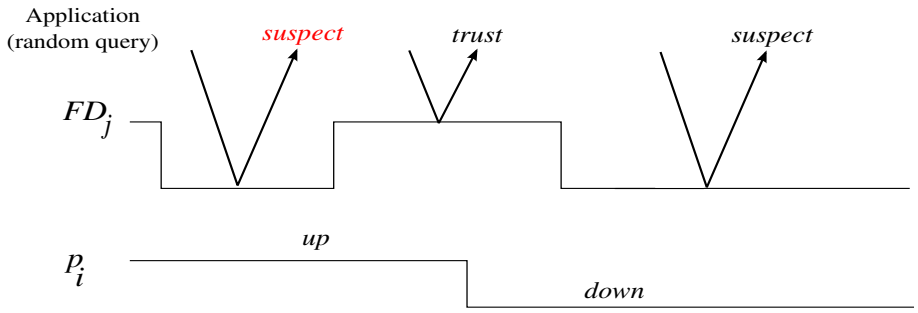


Figura 3.4: Representação da métrica P_A.

Os autores de [Chen et al., 2000] também propuseram um algoritmo para detecção de falhas que pode ser configurado com base nessas métricas e de acordo com o comportamento da rede, incluindo a probabilidade de mensagens perdidas (p_L), atraso ($E(D)$) e variância do atraso ($V(D)$) das mensagens de *heartbeat*.

A métrica P_A descrita no parágrafo anterior é utilizada neste trabalho para avaliar o serviço de detecção de falhas, ao passo que as métricas apresentadas na Figura 3.3 são utilizadas

como parâmetros de QoS fornecidos pelas aplicações, como requisitos que devem ser garantidos pelo detector de falhas. O comportamento da rede também é considerado, tanto para calcular o *timeout*, quanto para ajustar o detector de falhas de acordo com os parâmetros de QoS.

Vale ressaltar que uma das contribuições da implementação descrita no próximo capítulo do presente trabalho é utilizar as métricas descritas nesta seção como parâmetros de QoS, que podem ser requisitados por uma aplicação. Com estes parâmetros de QoS, o detector de falhas pode utilizar um mecanismo para ajustar o monitoramento dos processos, objetivando cumprir com as requisições recebidas. O mecanismo que ajusta o monitoramento dos processos utilizado neste trabalho, é baseado no algoritmo proposto por Chen, Toueg e Aguilera [Chen et al., 2000]. Vale ressaltar também que o algoritmo original proposto por Chen, Toueg e Aguilera pode ser considerado de difícil entendimento, tornando complexa sua reprodução em um ambiente real. Neste trabalho, uma contribuição é também desvendar o algoritmo original em um formato mais simples, facilitando a sua compreensão e consequente implementação. Esta proposta, bem como as adaptações realizadas no algoritmo, é apresentada no próximo capítulo.

3.2 Algoritmo de Consenso Paxos

Em sistemas distribuídos tolerantes a falhas são frequentes as situações em que os processos precisam entrar em um acordo referente a uma decisão a ser tomada. Como exemplo, pode-se citar transações atômicas como *commit* em banco de dados, difusão atômica, composição de grupo, entre outras. A realização do acordo distribuído não tem uma solução trivial, podendo-se afirmar que é um dos principais problemas em sistemas assíncronos tolerantes a falhas [Borran et al., 2012].

Em situações em que há a necessidade de se ter uma decisão em conjunto entre os processos participantes do sistema, um mecanismo fundamental para garantir a decisão unânime dos participantes é o consenso, que garante que todos os processos entram em comum acordo sobre um valor único com base em valores propostos inicialmente, considerando que esses valores iniciais podem ser diferentes para cada processo [Borran et al., 2012]. Portanto, em um protocolo de consenso, um conjunto de processos Π deve por unanimidade concordar numa decisão obtida. Esta decisão é relacionada ao grupo de valores inicialmente propostos por qualquer dispositivo $p_i \in \Pi$.

Paxos é um algoritmo de consenso tolerante a falhas projetado para replicação de máquina de estado [Lamport, 1998, Lamport, 2001]. O consenso está alicerçado em quatro propriedades, sendo uma propriedade de progressão (*liveness*) e três propriedades de segurança (*safety*): terminação, validade, integridade e acordo. Estas propriedades são descritas a seguir [Cachin et al., 2011]:

- *Terminação*: assegura que cada processo correto em algum momento decide por algum valor, sendo esta a propriedade que garante a progressão.
- *Validade*: se um processo decide por um valor v, então v foi proposto por algum processo.
- *Integridade*: garante que nenhum processo decide duas vezes.
- *Acordo*: dois processos corretos não decidem valores diferentes.

O algoritmo Paxos possui características importantes: garante a segurança do consenso em um sistema assíncrono sujeito a falhas e o progresso sob suposições de assincronia fraca. O algoritmo assume o modelo de falhas de parada com recuperação (*crash-recovery*). Os canais

são não-confiáveis, assim mensagens podem ser perdidas. No Paxos, os processos assumem os seguintes papéis distintos: *proposers*, *acceptors* e *learners*. Os *proposers* propõem um valor, os *acceptors* escolhem um valor e os *learners* aprendem o valor decidido. Um único processo pode assumir qualquer uma dessas funções e múltiplas funções simultaneamente. Paxos é ótimo em termos de resiliência [Lamport, 2006a]: para tolerar f falhas ele requer $2f + 1$ *acceptors* – isto é, para assegurar o progresso, um quórum de $f + 1$ *acceptors* devem estar sem-falha.

Para garantir que diversas execuções do consenso sejam realizadas, o algoritmo executa sequências separadas de instâncias do Paxos [Lamport, 2001]. Cada instância corresponde à execução do consenso e está associada a um valor decidido. Uma instância de Paxos executa em duas fases.

A Figura 3.5 apresenta um cenário de execução em duas fases com dois *proposers* (P_0 e P_1), três *acceptors* (A_0 , A_1 e A_2) e dois *learners* (L_0 e L_1). O *proposer* P_1 inicia o consenso ao receber um valor v através de uma requisição *propose*. Então, durante a primeira fase P_1 seleciona um número único de rodada e o envia em uma solicitação *prepare* aos *acceptors*. Ao receber uma solicitação *prepare* com um número de rodada maior que qualquer rodada que o *acceptor* tenha recebido previamente, o *acceptor* responde ao *proposer* prometendo que rejeitará qualquer requisição futura com números de rodada menores. Se o *acceptor* já aceitou um valor para instância atual (explicado a seguir), ele irá retornar esse valor ao *proposer* juntamente com o número de rodada recebido quando o valor foi aceito. Na Figura 3.5, a resposta dos *acceptors* é realizada em uma requisição *prepareResp* com os seguintes parâmetros: *n-rod* é o número de rodada, *v-rod* é a rodada em que o comando foi aceito e *v-accept* é o valor aceito. Caso não haja valor aceito, os dois últimos parâmetros são nulos. Quando o *proposer* recebe respostas de um quórum, ou seja, uma maioria de *acceptors*, o algoritmo segue para a segunda fase.

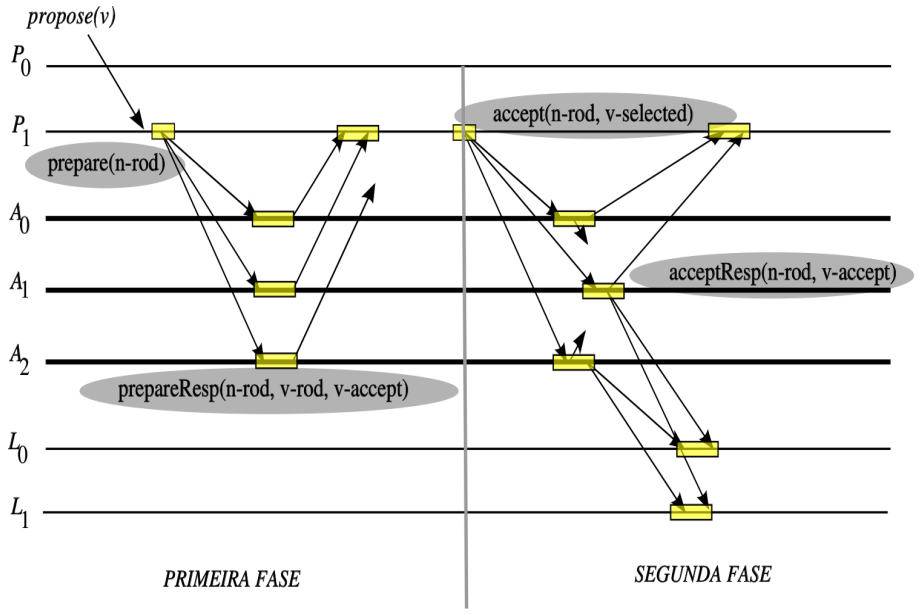


Figura 3.5: Paxos em duas fases (fonte [de Camargo e Duarte, 2017]).

Na segunda fase, o *proposer* seleciona um valor de acordo com a seguinte regra: se nenhum *acceptor* no quórum de respostas aceitou um valor, o *proposer* pode selecionar um novo valor para a instância (no caso, o valor v recebido na requisição *propose*). No entanto, se qualquer um dos *acceptors* retornou um valor na primeira fase, o *proposer* escolhe o valor com o maior número de rodada. O *proposer* então envia uma solicitação *accept* com o número de rodada

usado na primeira fase e o valor escolhido (*v-selected*) para os *acceptors*. Quando recebem essa solicitação, os *acceptors* enviam mensagens de confirmação (*acceptResp*) ao *proposer* e aos *learners*. Quando um quórum de *acceptors* aceita o valor, alcança-se o consenso.

Se múltiplos *proposers* simultaneamente executam o procedimento anterior para a mesma instância, então nenhum *proposer* pode conseguir executar as duas fases do protocolo e alcançar o consenso. Por exemplo, P_0 e P_1 na Figura 3.5 podem ficar competindo pelo maior número de rodada. Para evitar esse cenário onde *proposers* competem indefinidamente, um processo coordenador pode ser escolhido. Neste caso, *proposers* submetem valores ao coordenador, que então executa a primeira e a segunda fase do protocolo. Se o coordenador falha, outro processo assume a sua função. Paxos garante consistência apesar de *proposers* concorrentes e terminação na presença de um único coordenador.

3.3 Difusão Confiável

Em sistemas distribuídos, os processos utilizam um canal de comunicação para trocar mensagens. Um caso especial de comunicação ocorre quando um processo deseja transmitir uma mensagem que precisa ser entregue, garantidamente, a todos os demais processos corretos do sistema [Défago et al., 2004]. Essa comunicação é conhecida como difusão confiável (*reliable broadcast*) e é fundamental para a construção de aplicações distribuídas tolerantes a falhas [Pedone e Schiper, 2003].

A difusão confiável pode ser definida formalmente em termos de duas primitivas: *broadcast(m)* e *deliver(m)*, onde *m* é uma mensagem. A primitiva *broadcast(m)* é chamada para disseminar *m* para todos os processos. A primitiva *deliver(m)* significa que a mensagem *m* pode ser entregue para todos os processos corretos. O problema da difusão confiável concentra-se em garantir a entrega de *m* para todos os processos que fazem parte de um grupo, cuja comunicação ocorre exclusivamente por troca de mensagens [Ekwall e Schiper, 2007].

Há outras definições para as diferentes abstrações da difusão confiável [Guerraoui e Rodrigues, 2006]. Por exemplo, *best-effort broadcast* garante que todos os processos entregam o mesmo grupo de mensagens somente se o emissor da mensagem estiver correto (isto é, não falho). Além destas definições, um protocolo de difusão confiável pode ser definido informalmente em termos das seguintes propriedades [Eugster et al., 2004][Correia et al., 2006]:

- *Validade*: se um processo correto executa *broadcast(m)*, então ele após um tempo entregará *m*.
- *Acordo*: se um processo correto entregar uma mensagem *m*, então todos os processos corretos após um tempo entregarão *m*.
- *Integridade*: para qualquer mensagem *m*, cada processo correto entrega *m* no máximo uma vez e apenas se *m* foi previamente transmitida por um processo correto.

Note que *Integridade* garante que não haverá mensagens espúrias (que não foram geradas por processos do sistema), ao passo que *Validade* e *Acordo* garantem que uma mensagem transmitida (*broadcast(m)*) por um processo correto será entregue (*deliver(m)*) por todos os processos corretos [Hadzilacos e Toueg, 1994]. Assim, é importante perceber que se o processo transmissor de uma mensagem *m* falhar, a especificação da difusão confiável possibilita que a mensagem *m* seja entregue por todos os processos corretos ou por nenhum deles.

Dado que um processo emissor p_i seja capaz de realizar difusão confiável de *n* mensagens, é importante que um processo receptor p_j seja capaz de determinar a identidade do processo

p_i. Além disso, é necessário distinguir diferentes mensagens (m_1, m_2, ..., m_n) transmitidas pelo processo p_i. Assim, é importante considerar que cada mensagem m possua um identificador que determina o emissor, e um número de sequência que identifica a mensagem transmitida. Esses dois campos fazem cada mensagem m ser única no sistema.

3.3.1 Difusão Atômica

Enquanto as propriedades de difusão confiável suprem as necessidade de determinadas aplicações, para outras, elas podem não ser suficientes [Hadzilacos e Toueg, 1994]. Por exemplo, a difusão confiável não impõe nenhuma restrição quanto a ordem em que as mensagens são entregues. Neste caso, quando a entrega das mensagens precisa ser completamente ordenada é necessário utilizar um protocolo denominado de difusão atômica.

A difusão atômica requer que todos os processos corretos entreguem todas as mensagens na mesma ordem. Em geral, o algoritmo garante a ordem total com base em uma sequência de mensagens que é globalmente acordada. Por exemplo, é atribuído um número de sequência para cada mensagem a ser transmitida, onde cada uma delas deverá ser entregue para a aplicação respeitando a ordem acordada. Dessa forma, pode-se concluir que a difusão atômica é, na verdade, a difusão confiável que necessita satisfazer a seguinte condição [Hadzilacos e Toueg, 1994]:

- *Ordem total*: se dois processos corretos p e q entregam m e m', então p entrega m antes de m' se e somente se q entregar m antes de m'.

Por fim, a ordem total requerida pela difusão atômica implica que processos corretos após um tempo entregam a mesma sequência de mensagens.

3.3.2 Outras Propriedades de Ordenação

A difusão atômica apresentada na subseção anterior garante a ordenação das mensagens com base somente nos receptores, isto é, a especificação da propriedade é independente do emissor da mensagem [Défago et al., 2004]. Porém, pode-se ter mais restrições quando as propriedades estiverem relacionadas com os remetentes. Neste sentido, há duas propriedades descritas a seguir que são denominadas de *ordem FIFO* (*First-In/First-Out*) e de *ordem causal*.

Ordem FIFO

Segundo os autores em [Hadzilacos e Toueg, 1994], cada mensagem possui o seu contexto, sem o qual pode ser mal interpretada. Em algumas aplicações o contexto de uma mensagem consiste das mensagens previamente transmitidas pelo *emissor*. Por exemplo, em um sistema para reservas de passagens aéreas, o contexto de uma mensagem para o cancelamento de uma reserva consiste: a mensagem de cancelamento não pode ser entregue sem antes entregar a mensagem de reserva [Hadzilacos e Toueg, 1994]. Se as mensagens foram originadas pela mesma fonte, tal contexto requer a semântica de *difusão FIFO*.

A difusão FIFO é uma difusão confiável com a propriedade de ordenação FIFO, isto é:

- *Ordem FIFO*: se um processo transmite uma mensagem m antes de transmitir m', então nenhum processo correto entrega m' antes de entregar m.

Observe que a difusão atômica não garante que as mensagens serão entregues na ordem FIFO. Neste caso, se a aplicação necessitar de uma ordenação total respeitando a ordem FIFO, faz-se necessário a implementação da *difusão atômica FIFO*. Em outras palavras, *difusão*

atômica FIFO é uma difusão confiável que satisfaz as propriedades de *ordem FIFO* e de *ordem total*.

Ordem Causal

A ordem FIFO considera mensagens que são transmitidas pelo mesmo emissor. Por outro lado, uma mensagem m pode também depender das mensagens que o transmissor de m entrega, antes de transmitir m. Neste caso, a ordem FIFO não é mais suficiente. Por exemplo, em uma rede de notícias, se os usuários transmitem suas mensagens por difusão confiável e na ordem FIFO, o seguinte cenário pode ocorrer: usuário A transmite sua notícia, o usuário B entrega a notícia de A e responde com outra notícia que só pode ser entendida por usuários que tenham visualizado a notícia transmitida por A. O usuário C entrega a resposta de B antes de entregar a notícia transmitida por A; a resposta de B será mal interpretada.

No cenário descrito é necessário um tipo de ordenação que leva em consideração eventos com precedência causal. A noção de causalidade no contexto de sistemas distribuídos foi formalizada por Lamport [Lamport, 1978]. Ela é baseada na relação de "precedência" denotado pelo símbolo $\longrightarrow$. A relação de precedência é definida a seguir [Défago et al., 2004]:

- **Definição 1.** *Considere e_i e e_j sendo dois eventos em um sistema distribuído. A relação $e_i \longrightarrow e_j$ é garantida se uma das três condições descritas a seguir for satisfeita:*

  1. *e_i e e_j são eventos do mesmo processo, onde e_i ocorreu antes de e_j;*
  2. *e_i é a transmissão de uma mensagem m e e_j é o recebimento de m por outro processo; ou,*
  3. *Há um terceiro evento e_k, tal que, $e_i \longrightarrow e_k$ e $e_k \longrightarrow e_j$.*

Dada a relação de precedência causal (por exemplo, qualquer uma das três condições listadas acima) a difusão causal é uma difusão confiável que satisfaz a seguinte propriedade [Hadzilacos e Toueg, 1994]:

- *Ordem causal*: se a difusão de uma mensagem m preceder causalmente a difusão de uma mensagem m', então nenhum processo correto entrega m' antes de entregar m.

Por fim, se as aplicações necessitarem de uma ordenação total respeitando eventos com precedência causal, faz-se necessário a implementação da *difusão atômica causal*. Em outras palavras, *difusão atômica causal* é uma difusão confiável que satisfaz as propriedades de *ordem causal* e de *ordem total*. Destaca-se que a *ordem causal* é uma generalização da *ordem FIFO*. Na verdade, a propriedade da *ordem causal* é equivalente se combinarmos as propriedades da *ordem FIFO* com a seguinte propriedade [Hadzilacos e Toueg, 1994]:

- *Ordem local*: se um processo transmite uma mensagem m e algum processo entrega m antes de transmitir m', então nenhum processo correto entrega m' antes de entregar m.

Vale ressaltar que uma característica importante do sistema é garantir que a ordem a ser construída entre os processos seja realizada de maneira consistente. Neste trabalho é proposto um serviço que possibilita construir essa ordem de maneira consistente. O serviço proposto é denominado de *AnyBone* e possibilita garantir a entrega ordenada de mensagens. *AnyBone* será descrito no Capítulo 8.

3.4 Considerações Parciais

Neste capítulo foi apresentada uma introdução sobre detectores de falhas, descrevendo suas propriedades e detalhes para o monitoramento dos processos, em particular foram descritos dois algoritmos básicos denominados de *Push* e *Pull*. Além disso, foi apresentado um grupo de métricas que permite avaliar a qualidade do serviço oferecido pelos detectores de falhas. As métricas descritas são utilizadas no próximo capítulo para ajustar parâmetros do detector de falhas com base nos valores de QoS enviados pelas aplicações. Neste capítulo, também foi abordado o protocolo de consenso, tendo como foco o algoritmo Paxos utilizado para garantir, através de suas funcionalidades, uma decisão única sobre valores previamente propostos. Por fim, abordamos algoritmos de difusão confiável, difusão atômica, FIFO e causal. Os conceitos e algoritmos abordados neste capítulo são utilizados para a construção de sistemas distribuídos confiáveis.

Capítulo 4

Um Serviço para Detecção de Falhas com Configuração de Parâmetros de QoS

Neste capítulo é apresentado um serviço proposto para monitorar e detectar processos falhos na rede. O serviço para detecção de falhas proposto permite o monitoramento de processos tanto em redes locais como em múltiplos sistemas autônomos via Internet. O serviço proposto é denominado de IFDS (*Internet Failure Detection Service*) e possibilita o ajuste de seus parâmetros de monitoramento com base em requisições de qualidade de serviço especificadas pelas aplicações. Por exemplo, uma aplicação pode requisitar um limite de tempo para a detecção de uma falha, e o serviço ajusta os parâmetros de monitoramento do detector de falhas com referência nestes valores de entrada. Além disso, são propostas duas estratégias (η_{max} e η_{GCD}) que permitem compartilhar o serviço de detecção de falhas entre múltiplas aplicações.

Um algoritmo é proposto para receber os valores de QoS de entrada fornecidos pelas aplicações, bem como apresentar os cálculos realizados para o ajuste dos parâmetros de monitoramento do detector de falhas. Considerando a rede local, o monitoramento dos processos ocorre por troca de mensagens via agentes SNMP, ao passo que, em redes locais distintas via Internet a comunicação ocorre com o auxílio de *Web Services* (WS). As informações para monitoramento dos processos são armazenadas em uma MIB, responsável também por executar as atividades do próprio detector de falhas através de trocas de mensagens *heartbeat* (estilo *Push*).

Para apresentar as funcionalidades e características do serviço proposto, este capítulo está organizado da seguinte forma. Na Seção 4.1 é apresentado o modelo de sistema. Na Seção 4.2 é apresentado o processo para a configuração do *timeout* adaptativo utilizado pelo IFDS, como também estratégias para configurar o detector de falhas de acordo com as necessidades das aplicações. Na Seção 4.3 é descrita a arquitetura do IFDS e seus principais componentes. A MIB proposta neste trabalho é denominada de *fdMIB* (*failure detector MIB*) e sua estrutura é apresentada na Seção 4.4. Por fim, os experimentos e as considerações parciais são descritos na Seção 4.5 e na Seção 4.6, respectivamente.

4.1 Modelo de Sistema

Como apresentado no Capítulo 3, detectores de falhas foram propostos por Chandra e Toueg [Chandra e Toueg, 1996] e sua abstração possibilita encapsular o indeterminismo, permitindo garantir as propriedades do algoritmo de consenso, mesmo em sistemas assíncronos perante falhas por *crash*. Sendo assim, um detector de falhas tem por objetivo determinar os estados dos processos. Os estados indicam se um processo está ou não suspeito de acordo com o monitoramento realizado. O monitoramento ocorre por troca de mensagens através de canais

de comunicação em um sistema distribuído, respeitando um limite de tempo denominado de *timeout*. Em geral, para contextualizar o ambiente para a execução dos detectores de falhas, uma série de formalismos e hipóteses são consideradas.

Sendo assim, neste trabalho considera-se um sistema distribuído assíncrono composto por um conjunto Π de n processos/hosts, incrementado com detector de falhas [Chandra e Toueg, 1996]. Os processos falham por colapso (*crash*), ou seja, deixam de executar suas tarefas prematuramente. O monitoramento dos processos, em cada rede local, ocorre por um detector de falhas não confiável, o qual nunca falha. O monitoramento dos processos pelo detector de falhas resulta nos seguintes estados: *incorreto* ou *suspeito*, ocorre quando uma mensagem de *heartbeat* não é obtida dentro de um intervalo específico de tempo; *correto* ou *não suspeito*, ocorre quando a mensagem de *heartbeat* é recebida pelo detector de falhas dentro do intervalo de tempo estabelecido. A comunicação é realizada por troca de mensagens através de um canal confiável, isto é, o canal não cria, não duplica e não altera mensagens de controle.

4.2 IFDS: Configuração do Timeout e dos Parâmetros de QoS

O serviço para detecção de falhas proposto neste capítulo é descrito detalhadamente nesta seção. Inicialmente, é apresentado o processo para a configuração do *timeout* adaptativo implementado no IFDS (ver Seção 4.2.1). Na sequência, Seção 4.2.2, é descrita a estratégia utilizada para ajustar parâmetros de monitoramento do IFDS com referência nos valores de QoS fornecidos pelas aplicações e de acordo com as condições da rede. Em específico, a solução proposta pode ser utilizada para atender as requisições de uma única aplicação ou pode ser compartilhada por múltiplas aplicações. Para o compartilhamento do serviço, são propostas duas estratégias denominadas de η_{max} e η_{GCD}. Na estratégia η_{max} os parâmetros de monitoramento do detector são maximizados para contemplar as necessidades temporais de todos os processos. Na estratégia η_{GCD} calcula-se o máximo divisor comum dos valores computados com base nas requisições de QoS das aplicações, para então ajustar os parâmetros de monitoramento do detector de falhas.

4.2.1 Configuração do *Timeout* Adaptativo

Indiscutivelmente, uma importante função executada pelos detectores de falhas é o cálculo para determinar o intervalo de *timeout*. Por exemplo, um intervalo de *timeout* muito longo aumenta o tempo para a detecção de uma falha, um intervalo muito curto aumenta as chances para o detector de falhas cometer enganos. Dadas as características da Internet (em tempo e espaço), bem como as diferentes necessidades das aplicações, é impraticável configurar um valor fixo para o intervalo de *timeout*.

Neste contexto, uma importante abordagem utilizada para determinar o *timeout* é configurá-lo de maneira adaptativa, por exemplo, analisar as variações nos canais de comunicação e considerar as necessidades das aplicações para configurar o *timeout* com um valor que esteja de acordo com as condições do ambiente. A ideia é buscar estratégias que permitam prever o instante de tempo em que a próxima mensagem de monitoramento será recebida pelo detector de falhas.

Neste sentido, o serviço proposto por [Chen et al., 2000] (*Chen's FD*) estima-se o tempo de chegada da próxima mensagem de *heartbeat*, com base em um histórico de amostras de tempos passados. Este valor estimado é somado a uma margem de segurança que é constante.

No trabalho de [Bertier et al., 2003] (*Bertier's FD*), o serviço proposto também estima o tempo para a chegada da próxima mensagem de *heartbeat* (*Expected Arrival - EA*), mas diferentemente de *Chen's FD*, a margem de segurança somada ao valor do *timeout* é adaptativa (α). A adaptação da margem de segurança é baseada no algoritmo do protocolo TCP proposto por Jacobson [Jacobson, 1988]. EA é derivada de uma média das n mensagens de *heartbeat*, onde n é o tamanho do histórico das últimas mensagens recebidas.

O serviço para detecção de falhas proposto neste trabalho calcula o intervalo de *timeout* com base na estimativa do tempo de chegada da próxima mensagem de *heartbeat* somado a uma margem de segurança adaptativa. Considere dois processos: p_i e p_j, onde p_j monitora p_i. A cada intervalo de tempo η (lembrando que η representa a periodicidade de envio de mensagens), p_i envia uma mensagem de *heartbeat* para o monitor p_j. Considere m_1, m_2,...,m_k sendo as mensagens de *heartbeat*, em que k é a mais recente mensagem de *heartbeat* recebida por p_j. Sejam A_1, A_2,...,A_k os instantes de tempos nos quais as mensagens de *heartbeats* foram recebidas por p_j de acordo com o seu relógio local. EA pode ser estimado da seguinte forma:

$$EA_{k+1} = \frac{1}{k}\left(\sum_{i=1}^{k}(A_{(i)} - \eta)\right) + (k+1)\eta \tag{4.1}$$

EA_{k+1} é o instante de tempo estimado para a chegada da próxima mensagem (a mensagem m_{k+1}) de *heartbeat*. Sendo assim, o valor para o instante de tempo em que o próximo *timeout* (τ) irá expirar, é calculado da seguinte forma:

$$\tau_{(k+1)} = EA_{(k+1)} + \alpha_{(k+1)} \tag{4.2}$$

Para tornar a detecção de falhas mais precisa, a margem de segurança α é calculada de acordo com o algoritmo de Jacobson que trabalha com base nos instantes de tempo em que as mensagens de *heartbeat* (hb) são recebidas, isto é, $diff_{(k)}$ apresentado na Expressão 4.3, representa a diferença de instante de chegada entre a mensagem m_k e m_{k-1}:

$$diff_{(k)} = hb_k - hb_{k-1} \tag{4.3}$$

Na Expressão 4.4, $delay_{(k+1)}$ é o valor do atraso que é calculado em termos da diferença $diff_{(k)}$, e γ representa o peso das novas amostras, Jacobson sugere o valor de $\gamma = 0,1$:

$$delay_{(k+1)} = (1-\gamma) \cdot delay_{(k)} + \gamma \cdot diff_{(k)} \tag{4.4}$$

$var_{(k+1)}$ representa a variação da diferença:

$$var_{(k+1)} = (1-\gamma) \cdot var_{(k)} + \gamma \cdot (|diff_{(k)} - var_{(k)}|) \tag{4.5}$$

Por fim, a margem de segurança $\alpha_{(k+1)}$ é obtida através da Expressão 4.6, onde β e ϕ são constantes que recebem os seguintes valores: $\beta = 1$ e $\phi = 4$; como sugerido em [Bertier et al., 2003].

$$\alpha_{(k+1)} = \beta \cdot delay_{(k+1)} + \phi \cdot var_{(k+1)} \tag{4.6}$$

Com base nos cálculos apresentados nesta seção, o intervalo de *timeout* τ é ajustado a cada nova mensagem de *heartbeat* recebida pelo detector de falhas.

4.2.2 Configurando o Detector de Falhas com Base nos Parâmetros de QoS

Em detectores de falhas, um desafio é oferecer um serviço que possa ser adaptado de acordo com as necessidades de cada aplicação. Por exemplo, considerando os propósitos de cada aplicação, definir um período curto para o tempo de detecção de uma falha pode ser uma boa estratégia, mesmo que este serviço esteja sujeito a detectar falhas de forma incorreta. Por outro lado, para uma aplicação que tolera maiores atrasos, ter um serviço que comete menor número de erros pode ser melhor, mesmo que o tempo para a detecção da falha tenha um período mais longo. Portanto, é relevante oferecer um serviço capaz de se adaptar às necessidades de cada aplicação.

Considerando o exposto, nesta seção é apresentado um algoritmo que permite configurar o detector de falhas de acordo com as necessidades de QoS de cada aplicação. Inicialmente, as funcionalidades do algoritmo são apresentadas considerando requisições de uma única aplicação. Subsequentemente, o algoritmo é estendido para funcionar com duas estratégias (denominadas η_{max} e η_{GCD}) que trabalham de acordo com as requisições de QoS para múltiplas aplicações. Desta forma, mesmo se duas aplicações usarem o detector de falhas para monitorar o mesmo processo, a estratégia deve suportar diferentes requisições de QoS, e um único intervalo de *heartbeat* deve ser calculado, satisfazendo todas as requisições. Além disso, o tempo limite para detecção de uma falha pode ser diferente, uma aplicação pode necessitar de um tempo pequeno, enquanto que outra pode suportar um tempo maior. Se a mensagem de *heartbeat* chegar depois do menor tempo de detecção, mas antes do maior tempo, somente uma aplicação é informada sobre a falsa suspeita detectada.

Como ilustrado na Figura 4.1, uma aplicação pode explicitar suas necessidades de QoS enviando ao detector de falhas os seguintes parâmetros de entrada:

- T_D^U: um limite máximo para o tempo de detecção;
- T_M^U: um limite máximo para a duração de um erro;
- T_{MR}^L: um limite mínimo para a ocorrência entre dois erros consecutivos.

Com os parâmetros que identificam as necessidades de QoS da aplicação, o detector de falhas processa os dados de entrada e busca um valor adequado para η, de acordo com os procedimentos detalhados a seguir.

Ativando QoS: Uma Única Aplicação

A Figura 4.1 apresenta uma aplicação que envia seus parâmetros de QoS para que o detector de falhas encontre um valor apropriado para η.

Note que o detector de falhas é composto por dois módulos: *Estimator* e *Configurator*. Dada uma mensagem de *heartbeat* recebida de um processo monitorado, o *Estimator* calcula três parâmetros: probabilidade de mensagens perdidas (p_L), variância do atraso ($V(D)$) e estimativa do tempo da próxima mensagem de *heartbeat* (EA). A probabilidade de mensagens perdidas é encontrada calculando $p_L = L/k$, onde k é o número total de mensagens que foram enviadas e L é o número de mensagens perdidas. $V(D)$ é calculado observando a variância nos tempos das mensagens recebidas, enquanto EA é a estimativa do tempo de chegada da próxima mensagem de *heartbeat* calculado conforme mostrado na Expressão 4.1. O módulo *Configurator* é responsável por implementar e executar as funções do Algoritmo 1, em especial, a função do *Configurator* é tentar encontrar um valor para η para ser utilizado como parâmetro de monitoramento pelo detector de falhas.

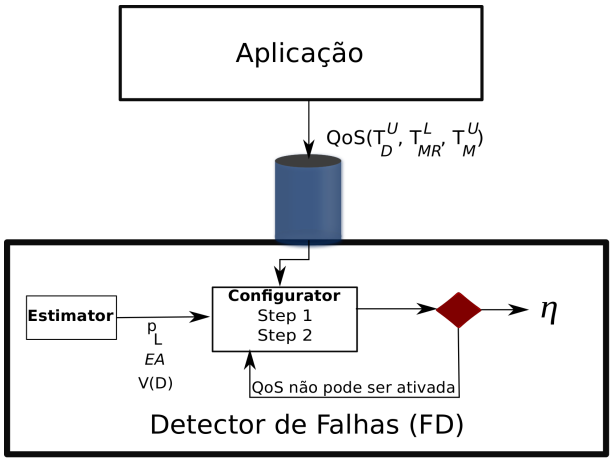


Figura 4.1: Calculando um valor apropriado para η.

Como apresentado na Figura 4.1 e explicado a seguir nesta seção, há situações em que os valores dos parâmetros requisitados pela aplicação não permitem a ativação de QoS (ver caso (i) e (ii) nesta seção). Os detalhes do funcionamento do *Step1* e *Step2*, ilustrados na Figura 4.1, são apresentados no Algoritmo 1. Vale ressaltar que o Algoritmo 1, que configura o detector para uma única aplicação, é adaptado com base no trabalho proposto por Chen [Chen et al., 2000]. Essa adaptação no algoritmo original refere-se à formatação/descrição do algoritmo, em particular tornar sua compreensão mais simples que sua versão original, a qual pode ser considerada não trivial por não incluir diversos detalhes relevantes.

O Algoritmo 1 é composto por dois passos: *Step1* e *Step2*.

O propósito do *Step1* é encontrar um limite superior para o intervalo de *heartbeat*, dada a entrada dos valores providos pela aplicação, isto é: (T_D^U, T_M^U, T_{MR}^L). No algoritmo, o limite superior é representado pelo símbolo η_{max}. No *Step1*, θ é calculado na linha 12, este valor deve refletir a probabilidade para a detecção de falhas que é baseada na variação do atraso $V(D)$ e no limite de tempo de detecção requerido T_D^U, lembrando que η_{max} deve representar um limite superior que considera o tempo de detecção e o tempo para a duração de um erro. Assim, η_{max} é configurado para ser o menor valor entre o limite máximo para o tempo de detecção (T_D^U) e a probabilidade para a detecção de falhas multiplicada pelo limite máximo para a duração de um erro ($\theta * T_M^U$). Este valor é obtido através da linha 16 ou da linha 18 do algoritmo.

Note que, se pelo menos um dos seguintes casos ocorrer: $\theta = 0$ ou $T_D^U = 0$ ou $T_M^U = 0$, então o máximo intervalo de *heartbeat* é zero; isto significa que não há um intervalo de *heartbeat* capaz de garantir as requisições de QoS. Nesta situação, *Step2* não é executado. Por outro lado, se no *Step1* $\eta_{max} \geq 0$, então a linha 19 do algoritmo é executada invocando o *Step2*.

Algorithm 1: Algoritmo para configuração de QoS.

// T_D^U: *limite máximo para o tempo de detecção*
T_M^U: *limite máximo para a duração de um erro*
T_{MR}^L: *limite mínimo para a ocorrência entre dois erros consecutivos*
θ: *reflete a probabilidade para detecção de falhas*
p_L: *probabilidade de mensagens perdidas*
$V(D)$: *variança do atraso*
η_{max}: *é configurado de acordo com o* $min(\theta * T_M^U, T_D^U)$
η: *intervalo de heartbeat* //

Step1 (T_D^U, T_M^U, T_{MR}^L) ▷ *no* **Step1** *calcula-se o valor para* η_{max}

$\theta := (1 - p_L)(T_D^U)^2/(V(D) + (T_D^U)^2)$;

if $((\theta > 0)$ *and* $(T_D^U > 0)$ *and* $(T_M^U > 0))$ **then**
    if $((\theta * T_M^U) > T_D^U)$ **then**
        $\eta_{max} := T_D^U$;
    else
        $\eta_{max} := \theta * T_M^U$;
    run Step2$(T_D^U, T_{MR}^L, \eta_{max})$
else
    return ("QoS não pode ser ativada");

Step2 $(T_D^U, T_{MR}^L, \eta_{max})$ ▷ *no* **Step2** *calcula-se o valor para* η

$\eta := \eta_{max}$; ▷ η_{max} *é utilizado como valor para inicializar* η
while $(f(\eta) < T_{MR}^L)$ **do**
    $\eta := \eta - (\eta * 0.01)$; ▷ *para* $f\eta$ *crescer o valor de* η *é reduzido*;
return (η); ▷ *dado de saída*

Function $f(\eta)$
return $(\eta * \prod_{j=1}^{\lceil T_D^U/\eta \rceil - 1} \frac{V(D)+(T_D^U - j\eta)^2}{V(D)+(p_L(T_D^U - j\eta)^2})$;

O objetivo do *Step2* é encontrar o menor valor de η que respeite o terceiro parâmetro de QoS (T^L_{MR}), sendo ele o limite mínimo para a ocorrência entre dois erros consecutivos. Como η deve ser maior do que zero e menor do que η_{max}, propomos inicializar η igual a η_{max} (linha 24). A cada novo valor definido para η, a seguinte condição é testada: $f_\eta \geq T^L_{MR}$ (linha 22). Enquanto esta condição não for verdadeira, f_η é recalculado e um novo teste é executado.

Para cada valor de η, f_η é calculado em sucessivas iterações. Note que quando η diminui f_η aumenta, o oposto também é verdadeiro. Por esta razão, sempre reduzimos o valor de η (linha 26, reduz em 1% o valor de η) até $f_\eta \geq T^L_{MR}$. Este fator de redução (1%) foi experimentalmente escolhido fornecendo um valor preciso, isto é, próximo do T^L_{MR} – se for definido um valor de 10% a precisão não é garantida, pode-se obter um valor aceitável mas muito distante de T^L_{MR}. Quando a condição $f_\eta \geq T^L_{MR}$ for garantida, o valor de η que satisfaz T^L_{MR} é finalmente encontrado.

Para ilustrar o funcionamento do algoritmo apresentado nesta seção, considere o seguinte exemplo:

Exemplo 1. Uma aplicação (App_1) necessita dos seguintes valores para os parâmetros de QoS: T^U_D=30s (uma falha é detectada no limite máximo de 30 segundos), T^U_M=60s (o detector de falhas corrige um erro no limite máximo de 60 segundos) e T^L_{MR}=432000s (o limite mínimo entre dois erros consecutivos é de 5 dias).

Além disso, assume-se que a probabilidade de mensagens perdidas é p_L=0.0 e a variância do atraso é $V(D)$=0.01. O algoritmo recebe os valores de entrada no *Step1* e calcula os seguintes valores: θ= 0.99 e η_{max}= min(30 , 59.99). Seguindo o algoritmo, no *Step2*, o valor de η é sempre reduzido até que a seguinte condição seja verdadeira: $f_\eta \geq 432000$. Para ativar esta condição, o algoritmo reduziu o valor de η 71 vezes, obtendo o seguinte resultado final η=14.6s. Neste momento, para assegurar os parâmetros de QoS requisitados pela App_1, o processo monitorado é configurado para enviar mensagens de *heartbeat* no período de 14.6s.

Vale ressaltar que, segundo [Chen et al., 2000], o valor encontrado para η resultante do *Step2* é o maior valor possível para η que permite cumprir com os parâmetros de QoS fornecidos pela aplicação, podendo ser definido qualquer outro valor menor, superior a zero. Em outras palavras, para o exemplo 1, η pode receber qualquer valor entre: $0 < \eta \leq 14.6$.

Ativando QoS: *Múltiplas Aplicações*

Quando várias aplicações utilizam o serviço de detecção de falhas, é possível que o IFDS seja configurado para atender a diferentes requisições de QoS. Em outras palavras, o IFDS pode ser compartilhado entre múltiplas aplicações. Sendo assim, é necessário encontrar um valor para o intervalo de *heartbeat* que satisfaça as requisições de QoS para todas as aplicações. Com essa abordagem, em cada intervalo de monitoramento, um único *heartbeat* é necessário para monitorar todos os processos. Isso permite também otimizar o número de mensagens de monitoramento.

Para esclarecer melhor como a estratégia proposta consegue economizar mensagens de monitoramento, considere um exemplo onde 5 aplicações desejam monitorar os estados de 100 processos. Para simplificar o exemplo, vamos assumir que todas as aplicações desejam o mesmo intervalo de *heartbeat*. Mesmo nestas condições, a cada intervalo de monitoramento o detector de falhas transmitiria 500 mensagens de *heartbeat*. Utilizando a estratégia de compartilhamento proposta, o número de mensagens reduz para somente 100.

Neste trabalho assumimos que diferentes aplicações podem necessitar de diferentes intervalos de *heartbeat*, para satisfazer as suas requisições de QoS. Para determinar um intervalo de *heartbeat* comum, são propostas duas diferentes estratégias, η_{max} e η_{GCD}, descritas a seguir.

Estratégia η_{max}: a ideia é encontrar um único valor para η que satisfaça todas as requisições de QoS. Para executar esta tarefa, no lugar de usar $\eta_{max} = min(\theta T_M^U, T_D^U)$ definido para uma única aplicação, é proposta para múltiplas aplicações a seguinte estratégia $\eta_{max} = min(\theta T_{M1}^U, T_{D1}^U, \theta T_{M2}^U, T_{D2}^U, ..., \theta T_{Mn}^U, T_{Dn}^U)$, onde $i = 1...n$, T_{Mi}^U e T_{Di}^U são as requisições de QoS para a aplicação i. Com esta modificação, é possível usar o Algoritmo 1 para calcular o valor apropriado de η.

Estratégia η_{GCD}: a ideia é calcular o máximo divisor comum (GCD - *Greatest Common Divisor*) entre todos os η_i, onde $i = 1...n$, e n é o número de requisições de QoS. Assim, para cada η_i um novo η_i' é encontrado da seguinte forma: $\eta_i' = 2^n$ onde $2^n < \eta_i$ e $\eta_i' \in \mathbb{Z}^+$. Então, o valor final é encontrado da seguinte maneira: $\eta_{GCD} = GCD(\eta_i', ..., \eta_n')$, onde $\eta_{GCD} > 0$, caso contrário esta estratégia não pode ser aplicada. Considerando que a estratégia η_{GCD} possa ser factível (cumprindo as condições descritas neste parágrafo), o valor encontrado nesta estratégia sempre será menor do que o valor da estratégia η_{max}. Neste sentido, esta estratégia resulta em um número maior de mensagens na rede, uma vez que $\eta_{GCD} < \eta_{max}$. Por outro lado, ela traz outros benefícios como, reduzir o tempo de detecção de falhas (T_D) e melhorar a probabilidade de uma resposta exata (P_A), conforme pode ser visto nos experimentos apresentados na Seção 4.5.

Note que a ideia está em considerar que, se um processo envia uma mensagem de *heartbeat* a cada x unidades de tempo, ele também pode enviar um *heartbeat* a cada y unidades de tempo, se x divide y. Além disso, vale ressaltar que ambas estratégias (η_{GCD} e η_{max}) utilizam o Algoritmo 1 apresentado nesta seção. Para ilustrar o funcionamento das estratégias η_{GCD} e η_{max}, o exemplo 2 descreve um cenário composto por duas aplicações (App_1 e App_2), cada uma com diferentes parâmetros de QoS e executando no mesmo serviço de detecção de falhas.

Exemplo 2. A App_1 possui os mesmos parâmetros de QoS descritos no Exemplo 1. A App_2 necessita dos seguintes valores para os parâmetros de QoS: T_D^U=15s (uma falha é detectada no limite máximo de 15 segundos), T_M^U=30s (o detector de falhas corrige um erro no limite máximo de 30 segundos) e T_{MR}^L=864000s (o limite mínimo entre dois erros consecutivos é de 10 dias). Além disso, assume-se que a probabilidade de mensagens perdidas é p_L=0.0 e a variância do atraso é $V(D)$=0.01. O algoritmo recebe as requisições de QoS da App_1, da App_2 e executa a estratégia η_{max}. O valor final de η_{max} é gerado com os parâmetros fornecidos pela App_2, ou seja, definido de acordo com a expressão que obtém o menor valor executado no *Step1* e apresentado a seguir: $\eta_{max} = min((30, 59.99)_{app_1}, (29.99, 15.00)_{app_2}) = 15.00$. Como resultado final, tem-se η=7.2s, este valor é utilizado para garantir as requisições de QoS tanto para App_1 quanto para App_2. Por outro lado, utilizando a estratégia η_{GCD}, tem-se os seguintes resultados: App_1: η_1= $14.6 \rightarrow \eta_1'$= 8; App_2: η_2= $7.2 \rightarrow \eta_2'$= 4; $\eta_{GCD} = GCD(4, 8) \rightarrow 4$. Portanto, tem-se η_{GCD} = 4s, obtendo um valor final menor para η se comparado a estratégia η_{max}.

Vale ressaltar que utilizando múltiplas aplicações, a proposta apresentada se adapta mesmo em condições em que ocorre violação de QoS causadas por atrasos na comunicação. Por exemplo, se o IFDS detectar que não é possível sustentar os valores mínimos de QoS requeridos, ele reajusta o valor de η aumentando para um outro valor pré-estabelecido, caso haja. Nesse caso o serviço garante, mesmo que parcial, a QoS requerida para determinadas aplicações. Por exemplo, considerando o exemplo 2, η pode ser reajustado para outro valor, desde que: $\eta \leq 14.6$.

4.3 IFDS: Arquitetura

Nesta seção, é descrita a arquitetura do serviço para detecção de falhas proposto neste capítulo. A Figura 4.2 mostra o IFDS monitorando processos localizados em duas redes locais distintas: LAN (*Local Area Network*) A e LAN B. Estas duas redes locais estão geograficamente

distantes e conectadas pela Internet. Em cada rede local há um *Host Monitor*, responsável por monitorar os processos através da configuração de um *timeout* τ, e um *Host Monitorado*, responsável por encaminhar mensagens de *heartbeat* ao *Host Monitor*, a cada intervalo de tempo η. As aplicações (*App*) podem se registrar no *Host Monitor* para receber informações sobre os estados dos processos.

Inicialmente, toda aplicação que precisa obter informações sobre os estados dos processos deve configurar seus parâmetros de QoS para que o IFDS ajuste o monitoramento de acordo com as informações de entrada fornecidas pela aplicação. O resultado deste processo é a definição do valor de η, como descrito na seção anterior. O *Host Monitor* disponibiliza a informação do η para que o *Host Monitorado* se ajuste, no sentido de enviar mensagens a cada η intervalo de tempo.

Como veremos na Seção 4.4, o protocolo SNMP é utilizado entre o *Monitor* e os *Hosts Monitorados* para trocar mensagens de monitoramento na rede local. Cada *Host Monitorado* é registrado em uma MIB. Se o *Host Monitorado* estiver localizado em outra rede, onde a comunicação ocorre através da Internet, emprega-se o uso de *Web Services* para a comunicação entre as duas redes. Neste caso, uma mensagem SNMP será encapsulada usando o protocolo SOAP (*Simple Object Access Protocol*).

Note que todo o monitoramento é realizado localmente em cada domínio. Monitores em diferentes domínios se comunicam para obter informações de estados de outros processos. Esta comunicação é implementada de duas maneiras. O *Monitor* ou uma aplicação local requisita a outro *Monitor* para obter as informações de estados dos processos ou, alternativamente, o host pode se registrar em um monitor remoto para receber notificações de troca de estado dos processos. Em ambos os casos emprega-se o uso de *Web Services*. Desta maneira, *Web Services* são utilizados como *gateways* SNMP para comunicar remotamente através da Internet. O uso de *Web Services* é transparente para a aplicação a qual emprega somente SNMP como interface de acesso ao *Host Monitor*.

As chamadas remotas via Internet são realizadas sempre que deseja-se obter o estado de um *Host Monitorado*, que está localizado em outra rede geograficamente distinta. Por exemplo, considere a Figura 4.2 onde a App_1, localizada na rede A, deseja obter o estado do processo que está localizado na rede B. Neste caso, a App_1 invoca uma chamada para o correspondente *Web Service*, que por sua vez executa uma operação SNMP na rede B. O agente SNMP acessa as informações dos processos na *fdMIB* localizada no *Host Monitor*. Por fim, o *Web Service* retorna as informações acessadas na *fdMIB* para a aplicação requisitante. A *fdMIB* exerce papel fundamental na arquitetura descrita na Figura 4.2 e os detalhes de sua implementação são apresentados na próxima seção.

4.4 Implementação da *fdMIB*

O protocolo SNMP é utilizado para a comunicação entre o *Host Monitor* e o *Host Monitorado*, as mensagens trocadas entre os *hosts* manipulam as informações armazenadas em uma MIB. A *fdMIB* proposta neste trabalho e os detalhes de seu funcionamento são apresentados nesta seção.

Na Figura 4.3 são ilustrados os parâmetros de QoS recebidos pela *fdMIB*. Os parâmetros são fornecidos pela aplicação através de um comando *snmpset*. O comando *snmpset* manipula os dados na *fdMIB*. Na Figura 4.3 as seguintes informações são fornecidas por parâmetro: endereço IP do *Host Monitor*, endereço da *fdMIB*, o identificador do objeto desejado (OID – *Object IDentifier*), o endereço de IP e porta de comunicação do *Host Monitorado* e, por fim, os valores de QoS, que podem ser fornecidos por uma única aplicação ou por várias aplicações, conforme

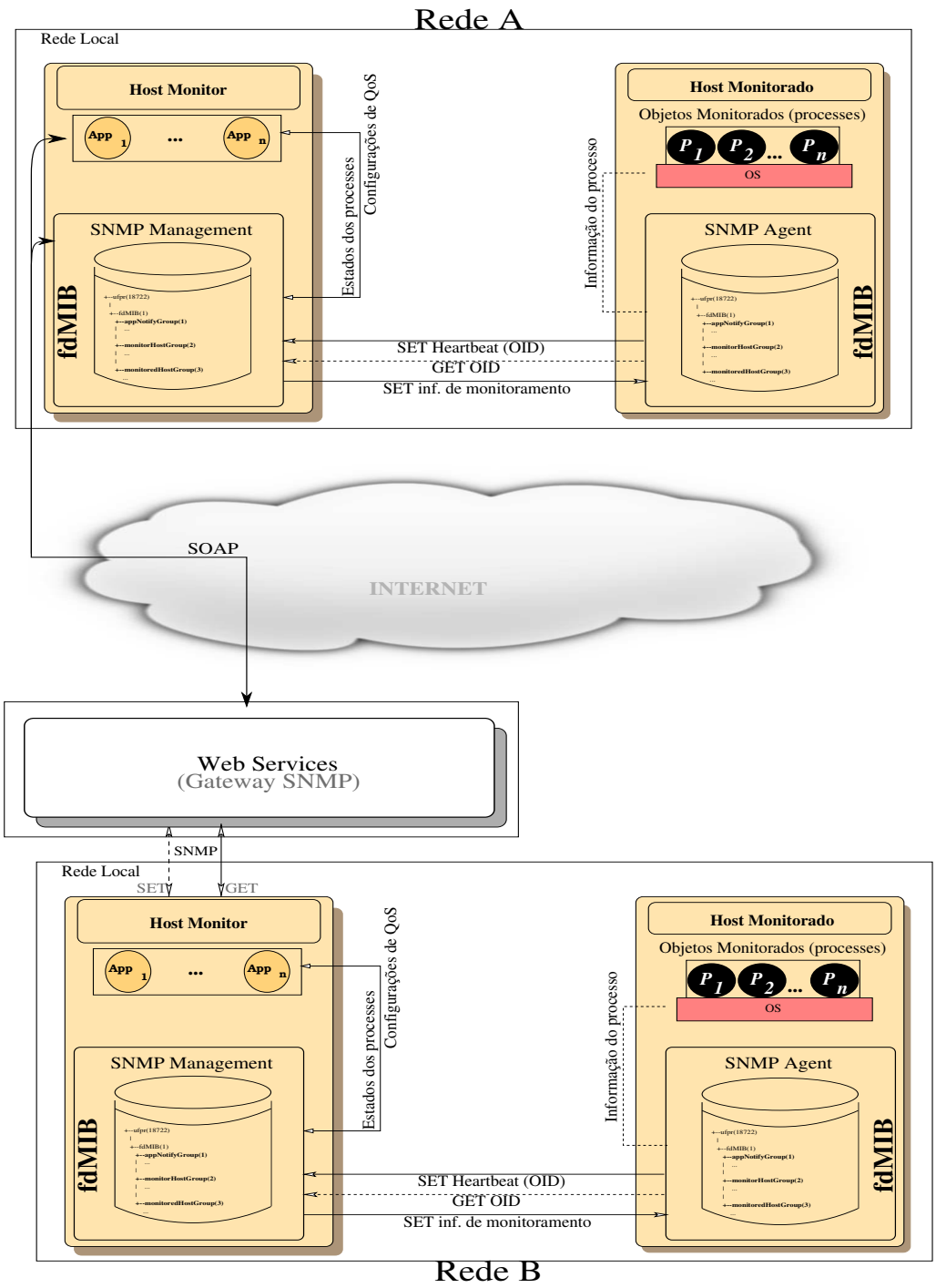


Figura 4.2: Arquitetura do IFDS.

apresentado na Figura 4.3. Além das informações de QoS, o *Estimator* é responsável por realizar cálculos estatísticos (p_L, $V(D)$ e EA) e o *Configurator* executa o *Step1* e *Step2* do Algoritmo 1. Estas tarefas são executadas pela *fdMIB* através da definição de três grupos: Grupo *Host Monitor*, Grupo *Host Monitorado* e Grupo *Notificar Aplicações*. Estes três grupos são formados por objetos que descrevem informações sobre todos os componentes da arquitetura, os detalhes destes grupos são apresentados na Figura 4.4 e descritos a seguir.

Grupo Notificar Aplicações (`appNotifyGroup`): este grupo é responsável por manter informações dos parâmetros de QoS fornecidos pelas aplicações. Cada aplicação que desejar receber informações sobre os estados dos processos deve informar, além do seu endereço

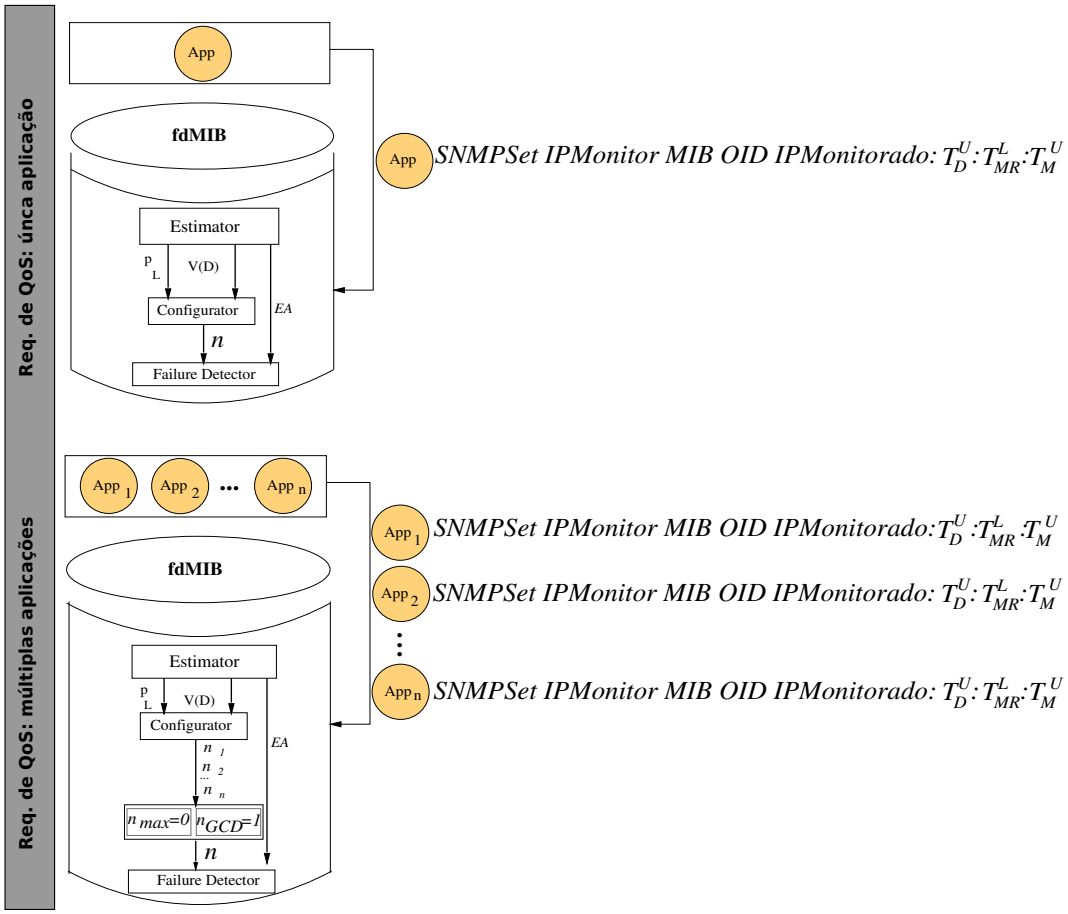


Figura 4.3: *fdMIB* recebe os valores de QoS e sempre calcula um único η para ser compartilhado entre todas as aplicações.

de IP e porta, os seguintes parâmetros: T_D^U, T_M^U e T_{MR}^L. Uma descrição sobre cada um destes parâmetros pode ser encontrada na Seção 3.1.2.

O `appNotifyGroup` possui um objeto denominado de `receiveHB` usado para controlar as mensagens de *heartbeat* recebidas dos processos monitorados. A cada mensagem de *heartbeat* recebida, a *fdMIB* atualiza o estado do processo correspondente. Neste momento, o objeto denominado de `statusChange` é atualizado para 0, indicando que o respectivo processo está correto, caso contrário, o valor será atualizado para 1, indicando que o processo está falho. Cada mensagem de monitoramento é identificada pelo seu OID (*Object IDentifier*), atualizando o valor do objeto correspondente. Por fim, para aplicações registradas na *fdMIB*, é possível enviar notificações de mudanças de estados dos processos via SNMP Traps (`notifyTrap`).

Grupo *Host Monitor* **(**`monitorHostGroup`**):** neste grupo são armazenadas informações sobre os processos monitorados, incluindo: endereço IP (`ipAddr`), número da porta para comunicação (`portNumber`) e estado do processo (`statusHost`), outras informações incluem: número de falsas suspeitas (`falseDetection`), intervalo de *heartbeat* (`redFreq`), cálculos estatísticos conforme apresentado na Figura 4.3 e parâmetros de QoS (`current_TD`, `current_TM` e `current_TMR`). Os parâmetros de QoS são continuamente checados para observar se algum parâmetro requisitado pela aplicação tem sido violado (por exemplo, `current_TD` > `TD_U`). No `monitorHostGroup` são gerenciadas todas as atividades de monitoramento, em particular o gerenciamento e controle das *threads* responsáveis por implementarem o *timeout* para cada processo monitorado.

Grupo *Host Monitorado* (`monitoredHostGroup`)**:** este grupo é responsável por enviar periodicamente as mensagens de *heartbeat* ao *Host Monitor* sendo, portanto, executado pelo *Host Monitorado*. Com a finalidade de se comunicar com o *Host Monitor*, as seguintes informações são mantidas pelos objetos: endereço IP do monitor, OID do próprio (o OID é usado para identificar o *Host Monitorado* na MIB) e o intervalo de *heartbeat*. Vale lembrar que η é calculado no *Host Monitor* de acordo com as requisições de QoS, para então ser enviado ao *Host Monitorado*, de forma a configurá-lo para encaminhar mensagens na periodicidade indicada.

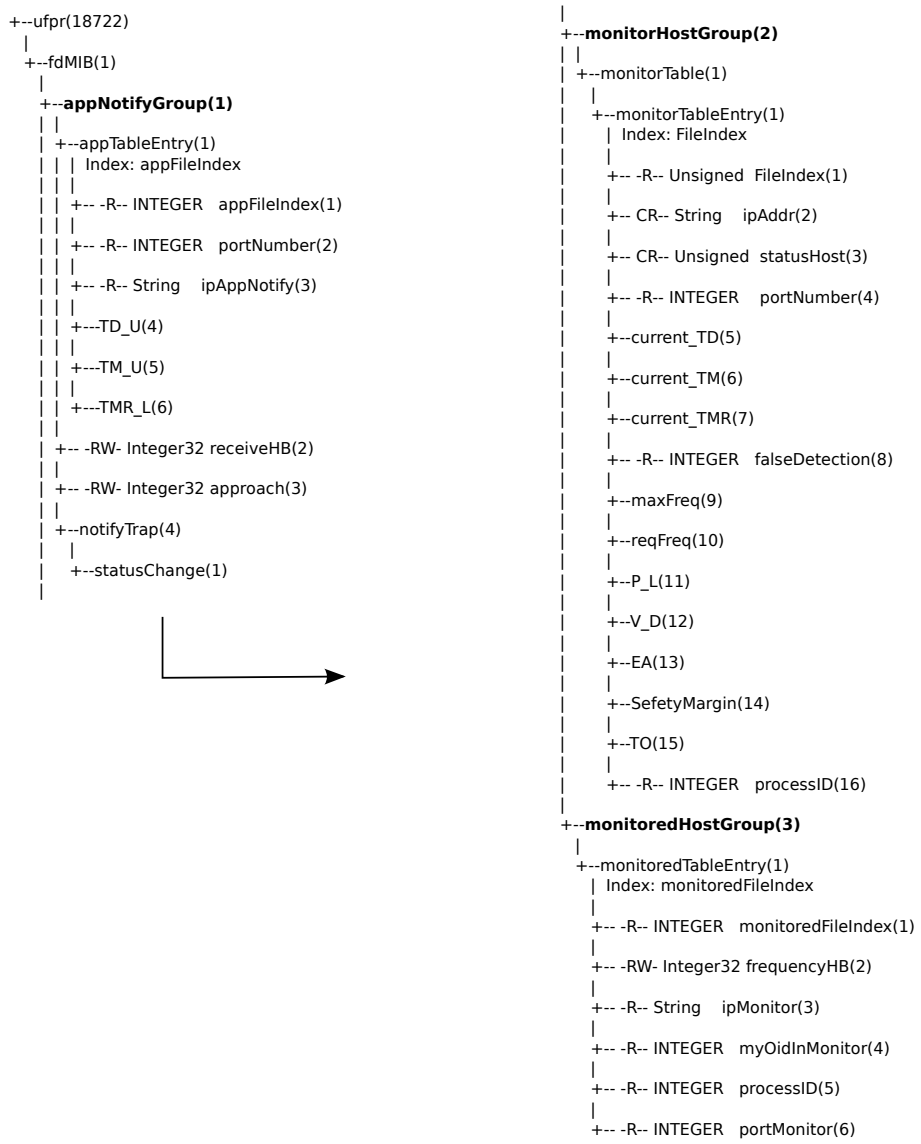


Figura 4.4: Estrutura dos grupos e objetos que compõem a *fdMIB*.

4.5 Avaliação Experimental do IFDS

Nesta seção são apresentados experimentos executados com o objetivo de avaliar o serviço para detecção de falhas proposto. Os experimentos foram executados em uma rede local e através de múltiplos sistemas autônomos via Internet. Os experimentos na rede local foram executados no laboratório Larsis da UFPR (Universidade Federal do Paraná), ao passo

que os experimentos via Internet foram executados no ambiente do PlanetLab[1]. Três aplicações independentes são utilizadas, cada uma com diferentes parâmetros de QoS como mostra a Tabela 4.1.

Tabela 4.1: Parâmetros de QoS de cada aplicação.

Aplicações	T_D^U **(ms)**	T_M^U **(ms)**	T_{MR}^L **(ms)**
App_1	8000	60000	2592000000
App_2	14000	120000	2592000000
App_3	16000	240000	2592000000

Os parâmetros de QoS apresentados na Tabela 4.1 são similares àqueles usados nos exemplos apresentados por [Chen et al., 2000]. Por exemplo, a App_1 possui os seguintes requisitos de QoS: uma falha deve ser detectada no período máximo de 8 segundos (T_D^U=8000ms), o detector de falhas corrige um erro no limite máximo de 1 minuto (T_M^U=60000ms) e o detector de falhas comete no máximo 1 erro por mês (T_{MR}^L=2592000000ms).

A configuração do IFDS pode ser realizada usando o comando *snmpset*; por exemplo, o comando a seguir é executado para configurar a tabela do `monitorHostGroup` com os valores de QoS desejados (Figura 4.4). Neste exemplo, App_1 monitora o processo em um host com o endereço de IP 192.168.1.1, porta 80 e os parâmetros de QoS são mostrados na Tabela 4.1.

snmpset -v1 -c private localhost .1.3.6.1.4.1.18722 .1.2.1.1.2.1 s 192.168.1.1:80:8000:60000:2592000000

O comando `snmpset` mostrado é usado para atualizar os valores de um objeto SNMP. Este comando usa os seguintes parâmetros: endereço do host onde a MIB está sendo executada (*localhost*), o *id* do objeto a ser atualizado (.1.3.6.1.4.1.18722.1.2.1.1.2.1) e os dados que serão escritos no objeto (192.168.1.1:8:80:60:2592000000).

Com base no Algoritmo 1 (apresentado na Seção 4.2.2), nos dados de entrada apresentados na Tabela 4.1 e considerando as estratégias para o cômputo do η, tem-se os seguintes resultados:
$\eta_{max} = min(1.954467, 3.901890, 4.694764) \rightarrow \eta$=1.95 e
$\eta_{GCD} = GCD(1,2,4) \rightarrow \eta$=1.

Os valores encontrados de η são suficientes para atender as requisições de QoS para as três aplicações.

4.5.1 Resultados Experimentais: LAN

Para os experimentos executados na rede local, foram utilizadas duas máquinas físicas distintas com as seguintes configurações: processador Intel Core i5 CPU 2.50GHz com 4 núcleos e sistema operacional Ubuntu 12.04.4 executando a *fdMIB* como monitor, o *host* monitorado possui um processador Intel Core i5 CPU 3.20GHz com 4 núcleos executando o sistema operacional Ubuntu 13.10, com kernel 3.2.0-58 em ambas as máquinas. O canal de comunicação na rede local é Ethernet 100Mbit/s. A MIB foi projetada com o pacote Net-SNMP[2] versão 5.4.4.

Nos gráficos das Figuras 4.5, 4.6(a) e 4.6(b) são ilustrados os envios de 200 mensagens de *heartbeat* que ocorrem entre o monitor e o *host* monitorado. Durante as primeiras 50

[1]http://www.planet-lab.org
[2]http://www.net-snmp.org

mensagens, o intervalo de *heartbeat* é configurado igual a 1000ms. Este valor justifica-se por ser um valor que contempla tanto a estratégia η_{max}=1.95s, quanto a η_{GCD}=1s. Após o envio de 50 mensagens, o intervalo de *heartbeat* é alterado para 5000ms. Novamente, este valor justifica-se por não contemplar nenhuma das estratégias, cujos máximos são: $\eta_{max} \leq$ 4.69s e $\eta_{GCD} \leq$ 4s. Além disso, essa mudança no valor de η é para verificar o comportamento do serviço de detecção de falhas através da simulação de sobrecarga nos processos ou canais de comunicação (por exemplo, quando o intervalo de *heartbeat* passa de 1000ms para 5000ms, a razão desse atraso poderia ser causada por alguma sobrecarga), como também o comportamento do IFDS considerando os parâmetros de QoS.

Para calcular o EA, o tamanho da janela deslizante é definido para 5 mensagens. Pode-se observar na Figura 4.5 que há um tempo inicial para a estabilização do *timeout*, isso ocorre durante as primeiras 10 mensagens de *heartbeat*. Este mesmo cenário pode ser observado quando ocorre a mudança no intervalo de emissão de η=1000ms para η=5000ms, as curvas do 'EA' e do '*Timeout*' mostram que o IFDS leva um pequeno período para se adaptar a nova frequência de *heartbeat* (curva 'Heartbeat Freq'). Note que nesta transição a curva do '*Timeout*' permanece por um pequeno período abaixo da curva do 'Heartbeat Freq': durante este tempo o IFDS comete falsas suspeitas. Depois disso, o *timeout* cresce o suficiente para corrigir os erros cometidos.

Este mesmo experimento é utilizado nas Figuras 4.6(a) e 4.6(b), onde o objetivo é mostrar que as requisições de QoS das aplicações não foram violadas. Além do 'EA', 'Heartbeat Freq' e do 'RTT', também aparecem na Figura 4.6 o tempo de detecção ('TD'), o tempo de duração de um erro ('TM'), o tempo para a recorrência de um erro ('TMR') e os tempos de detecção de QoS requeridos pelas aplicações.

Na Figura 4.6(a) é possível observar que ocorrem 6 falsas detecções, estas ocorrências de falhas podem ser observadas através do parâmetro de 'TM'. A Figura 4.6(a) também apresenta as 3 aplicações com seus respectivos parâmetros de QoS (vide Tabela 4.1), onde cada aplicação especifica seu próprio T_D^U. Com base nestes parâmetros, é possível observar, na Figura 4.6(a), que o tempo T_D calculado nos experimentos é sempre menor do que T_D^U especificado pela App_1, T_D^U da App_2 e T_D^U especificado pela App_3. Por esta razão, as falsas suspeitas detectadas e mostradas na Figura 4.6(a), não são notificadas para as aplicações, mesmo quando η atinge o valor de 5.

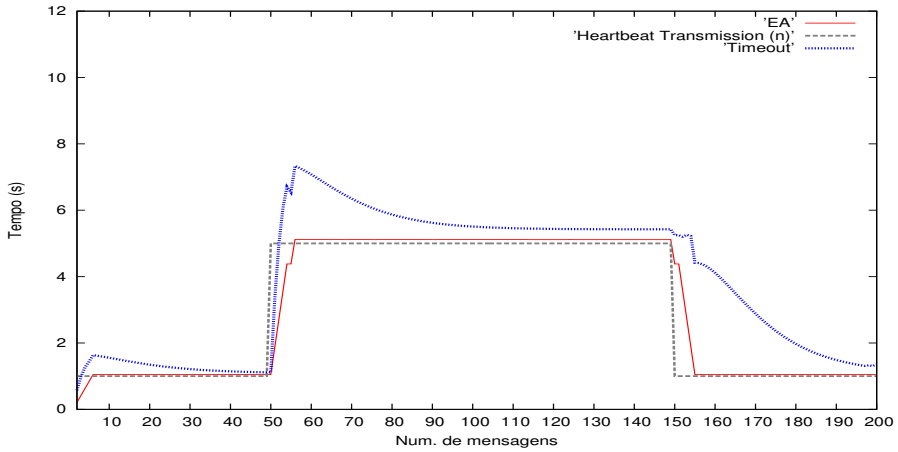
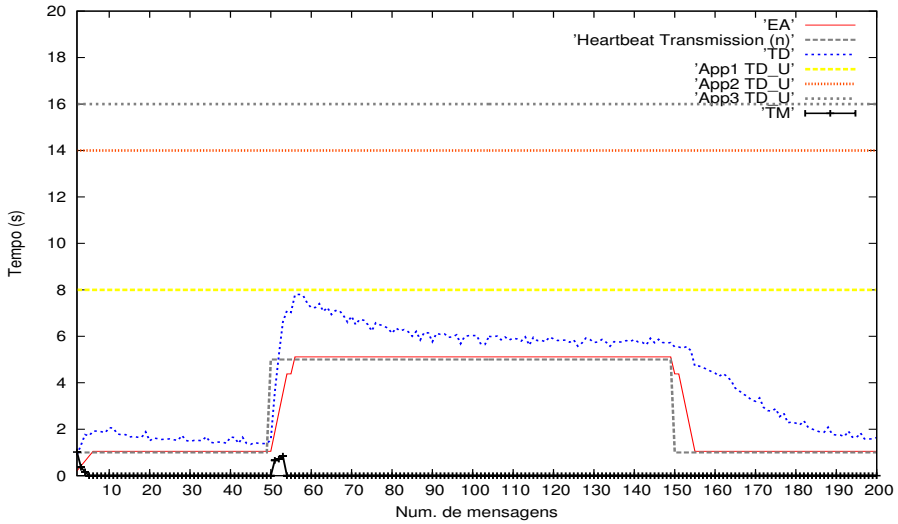


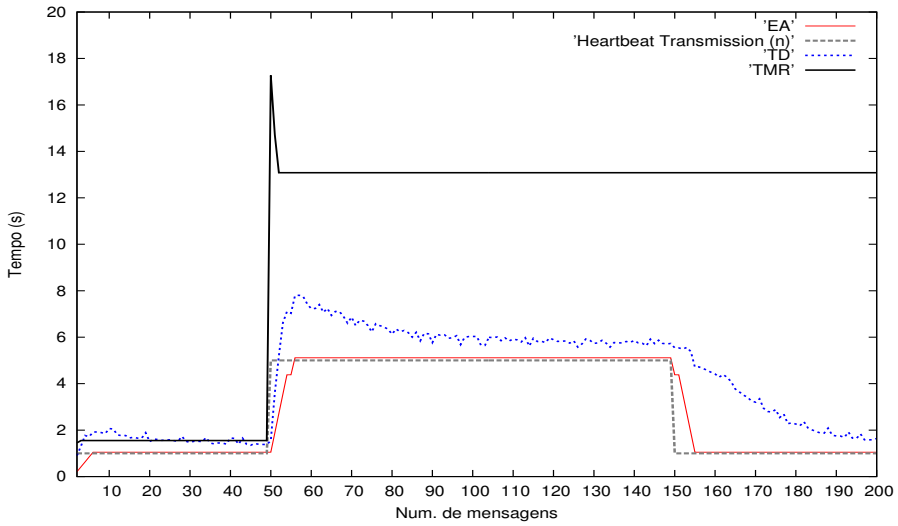
Figura 4.5: *Timeout* adaptativo com diferentes intervalos de emissão de mensagens de *heartbeat*.

Uma situação similar ocorre na Figura 4.6(b), onde a média do T_{MR} é de 13086ms, considerando as falsas detecções, de acordo com as requisições de QoS, T_{MR} deveria ser maior ou igual ao valor especificado pelas aplicações, isto é, T_{MR}^L=2592000000ms. Entretanto, como foi observado na Figura 4.6(a), nenhuma das 6 falsas suspeitas foram reportadas para as aplicações

($T_D \leq \min({T_D}^U{}_1, {T_D}^U{}_2, {T_D}^U{}_3)$), por esta razão, o valor do T_{MR} é 0 e não existem parâmetros de QoS que foram violados.



(a) T_D (Tempo de detecção) e T_M (Tempo de duração de um erro).



(b) T_{MR} (Tempo para recorrência ao erro).

Figura 4.6: Os parâmetros de QoS não são violados.

Para comparar as estratégias η_{max} e η_{GCD} propostas neste trabalho, utiliza-se os parâmetros apresentados na Tabela 4.1, onde o resultado encontrado para η foi: 1.95s e 1s, respectivamente. A Tabela 4.2 apresenta um comparativo entre as estratégias η_{max} e η_{GCD}. Para executar este experimento, é utilizada a métrica P_A descrita na Seção 3.1.2. A métrica P_A pode ser calculada da seguinte forma:

$$P_A = 1 - \frac{E(T_M)}{E(T_{MR})} \tag{4.7}$$

O experimento mostrado na Tabela 4.2 foi executado no período de 60 min, esta tabela mostra: o intervalo de *heartbeat* η (em segundos), o tempo de detecção T_D, a duração de um erro T_M, o tempo para a recorrência de erros T_{MR} e a métrica P_A descrita. Neste experimento duas falhas foram simuladas através da omissão de duas mensagens de *heartbeat*, forçando o estouro

do *timeout*. É possível verificar que η_{max} é melhor em termos de número de mensagens e mais apropriada para aplicações que não necessitam de um pequeno tempo de detecção. Por esta razão, η_{max} é mais apropriada para o monitoramento via Internet onde, em geral, as aplicações são mais tolerantes a atrasos. Como pode ser visto na Tabela 4.2, a estratégia η_{GCD} apresenta um menor tempo de detecção e melhor resultado para a métrica P_A. Portanto, sendo mais apropriada para o monitoramento em redes locais onde as aplicações geralmente não toleram períodos longos de atraso.

Tabela 4.2: Comparativo entre as duas estratégias: η_{max} and η_{GCD}.

	η (s)	T_D (ms)	T_M (ms)	T_{MR} (ms)	P_A	Num. de mensagens HB	Num. de falsas detecções
η_{max}	1.95	2458	1770	61190	0.9711	1754	2
η_{GCD}	1.00	1320	690	60592	0.9998	3551	2

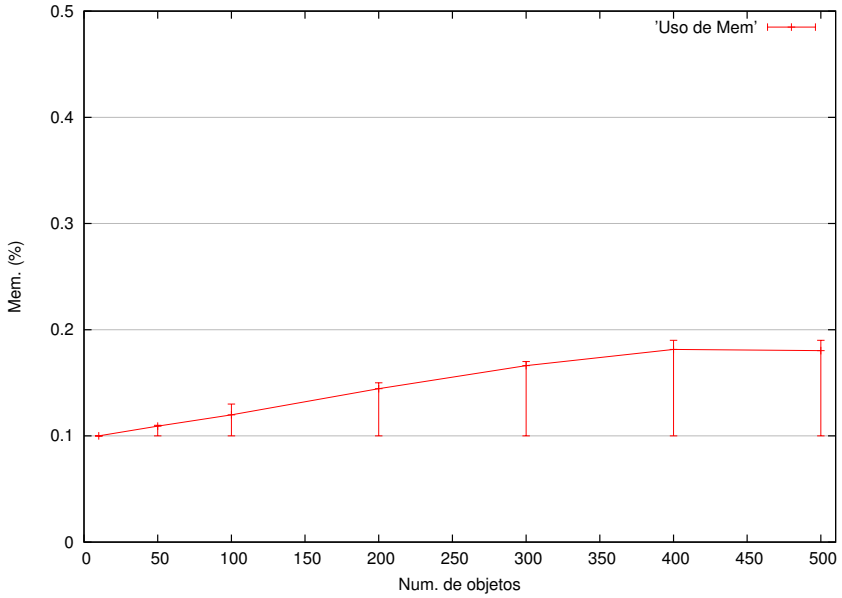
No próximo experimento são avaliados os recursos computacionais utilizados pelo IFDS, em especial, a Figura 4.7 mostra a utilização de CPU e memória no contexto do *Host Monitor*. Para sobrecarregar o uso destes recursos pelo serviço de monitoramento do detector de falhas, os experimentos foram executados com diferente número de objetos na MIB, sendo que cada objeto corresponde a um processo monitorado, e para cada processo monitorado são escalonadas, periodicamente, *threads* responsáveis pelo controle do *timeout*. Um único *host* físico é responsável por enviar mensagens de *heartbeat*, identificando cada um dos processos monitorados inclusos na MIB. Os experimentos envolvem a inclusão de cada um dos objetos na *fdMIB* e o recebimento das mensagens de *heartbeat*, o intervalo de envio de *heartbeat* é de 1ms. Para cada medida, 10 amostras foram coletadas e o serviço foi executado em um período de 60 min, variando o número de objetos para monitoramento. O uso da memória ilustrado na Figura 4.7(a), cresce quase linearmente até 400 objetos, havendo uma estabilização na utilização de memória, não chegando a atingir 0.2%.

Quanto à utilização de CPU, pode-se observar na Figura 4.7(b) que cresce quando o número de objetos aumenta até 100, a partir deste momento a utilização permanece quase constante, isto é, aproximadamente 7%. Essa utilização pode ser considerada baixa, quando comparada com a implementação do detector de falhas baseada em SNMP proposta por Defago [Wiesmann et al., 2006], onde a utilização de CPU atinge 11% com o mesmo intervalo de envio de *heartbeat* (1ms). Pode-se observar também que os valores máximos para a utilização de CPU apresentam um crescimento quase linear, estes picos são resultantes do registro inicial dos objetos na *fdMIB*. Entretanto, vale ressaltar que o procedimento para o registro de um processo na *fdMIB* é necessário, somente, na primeira vez em que for monitorado.

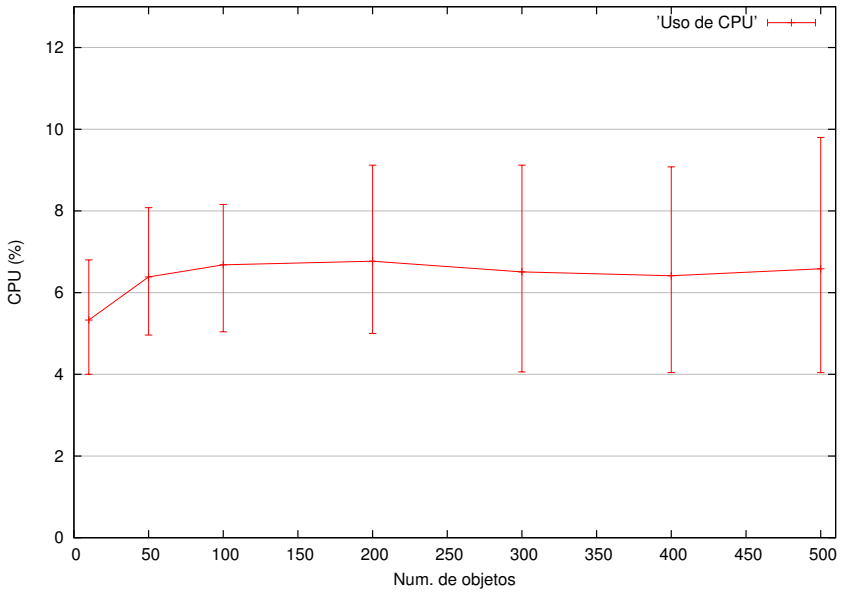
4.5.2 Resultados Experimentais: PlanetLab

O último experimento foi executado para investigar o *overhead* de comunicação quado são utilizados *Web Services* através da Internet. Os experimentos mostrados no gráfico da Figura 4.8 foram executados no PlanetLab. Os experimentos mostram o tempo para a notificação de uma falha considerando *hosts* espalhados em 5 continentes: América do Sul (Brasil), América do Norte (Canadá), Europa (Itália), Ásia (Rússia) e Oceania (Nova Zelândia).

As informações trocadas pelos *hosts* no PlanetLab ocorrem por duas maneiras: utilizando agentes SNMP e via *Web Services*. O tempo para a notificação da falha é computado usando um *Monitor* e *Monitorado Host* localizados no Brasil. O IFDS detecta a falha e notifica os demais, medindo o tempo percorrido desde a ocorrência da falha até o tempo em que ela é notificada. Em outras palavras, o tempo de notificação corresponde ao intervalo de tempo para a



(a) Utilização de memória.



(b) Utilização de CPU.

Figura 4.7: Avaliação de desempenho da *fdMIB* com o aumento de objetos para monitoramento.

detecção da falha somado ao intervalo de tempo para o conhecimento desta falha por um dado processo.

O gráfico da Figura 4.8 compara o tempo de notificação de uma falha usando agentes SNMP e *Web Services*. Como esperado, para todos os experimentos o tempo de notificação utilizando *Web Services* é maior, se comparado ao uso do protocolo SNMP. O acréscimo médio observado considerando o tempo de notificação da falha quando utilizando *Web Services* foi de 14,44%. A maior diferença ocorreu nos experimentos considerando a Nova Zelândia, sendo de aproximadamente 21% quando utilizando *Web Services*. Mesmo considerando o acréscimo para a notificação da falha, acreditamos que o acréscimo médio de 14,44% é razoável se considerarmos que a utilização de *Web Services* traz benefícios em termos de facilitar a comunicação em

diferentes domínios administrativos. Em outras palavras, a porta 80 utilizada para comunicação, em geral não é bloqueada pelos operadores dos domínios.

A Figura 4.8 mostra que o tempo de notificação respeita a distância física entre os *hosts*, por exemplo, o processo executando no Brasil, por estar próximo do detector de falhas, apresenta um tempo de notificação de aproximadamente 1,31s, considerando a comunicação via *Web Services*. Conforme a distância física entre os *hosts* aumenta, o tempo de notificação também aumenta, atingindo aproximadamente 2,01s para o *host* localizado na Nova Zelândia e considerando comunicação via *Web Services*. A partir destas observações conclui-se que uma aplicação poderia levar também em consideração a localização para então determinar o nível de QoS que pode ser obtida a partir de cada processo – por exemplo, o tempo de detecção de um processo que está próximo, será provavelmente mais baixo do que aquele que está geograficamente mais distante.

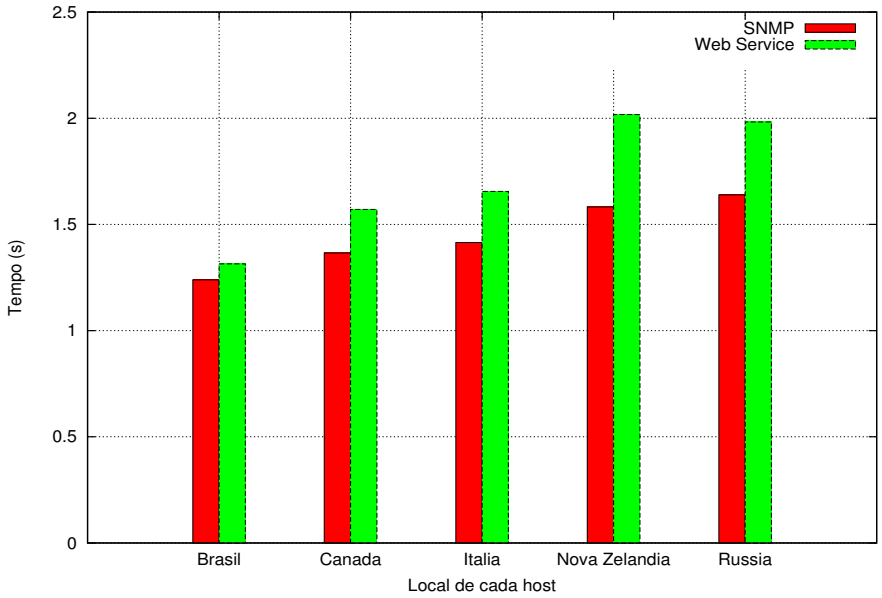


Figura 4.8: Tempo para notificação de uma falha, usando SNMP e WS.

4.6 Considerações Parciais

Neste capítulo foi proposto um serviço para detecção de falhas para processos executando em diferentes sistemas autônomos na Internet. O serviço proposto é denominado de IFDS e provê diversas funcionalidades implementadas através de uma arquitetura composta por agentes SNMP, *Web Services* e uma MIB denominada de *fdMIB*. A *fdMIB* executa as ações de monitoramento dos estados dos processos com referência nos parâmetros de QoS que são fornecidos pelas aplicações. Para garantir estes parâmetros de QoS, foram propostas duas estratégias (η_{max} e η_{GCD}) que permitem deduzir o valor do intervalo de *heartbeat* (η) com base nas informações de QoS fornecidas para o IFDS. Neste sentido, observou-se que a estratégia η_{max} é mais apropriada para aplicações que toleram maiores atrasos na comunicação (caso da Internet), ao passo que a estratégia η_{GCD} é indicada para uso em redes locais, obtendo maior probabilidade para uma resposta exata (P_A) e menor tempo para a detecção de uma falha.

Os resultados experimentais mostram que, quando os parâmetros de QoS não são violados, eventuais erros (e.g., falsas detecções) cometidos pelo detector de falhas, podem ser

omitidos para as aplicações. Foi mostrada também a eficiência da estratégia utilizada para a adaptação do *timeout* em diferentes situações de atrasos de comunicação. Por fim, mesmo considerando os atrasos adicionados pelo uso de *Web Services*, os experimentos executados no PlanetLab demonstraram a viabilidade de usar o IFDS para as comunicações entre diferentes sistemas autônomos na Internet.

Capítulo 5

Implementação de uma Função Virtualizada de Rede para Detecção de Falhas

Neste capítulo é apresentada a NFV-FD (*Network Function Virtualization - based Failure Detector*): uma função de rede para dar suporte à tolerância a falhas em aplicações distribuídas. A NFV-FD executa as funções de um detector de falhas não confiável descrito na Seção 3.1, e utiliza os benefícios de uma rede SDN (*Software Defined Network*) através do compartilhamento de informações fornecidas por um controlador da rede. O compartilhamento destas informações auxilia na obtenção dos atributos dos processos a serem monitorados, bem como permite determinar o próprio estado destes processos. Além disso, são também obtidas informações de monitoramento dos estados dos enlaces de comunicação, a partir dos próprios dispositivos de rede (*switches*).

A NFV-FD trabalha na formação de uma visão que indica quais processos estão suspeitos de terem falhado. Esta visão dos estados pode ser consultada por uma aplicação distribuída responsável por tomar decisões. Em outras palavras, a NFV-FD possibilita a execução de algoritmos distribuídos tolerantes a falhas. Um algoritmo para a difusão confiável (ver Seção 3.3) foi utilizado como estudo de caso. O trabalho ainda apresenta e detalha questões sobre a implementação da NFV. Em especial, discutimos, apresentamos resultados e implementamos a NFV-FD de diferentes maneiras: dentro e fora do controlador SDN, utilizando máquinas virtuais tradicionais e hospedada em *containers*. O desempenho do serviço de detecção de falhas e o impacto da NFV na utilização dos recursos para seu processamento, são também avaliados neste trabalho.

Este capítulo está organizado da seguinte forma. Na Seção 5.1 é apresentado o modelo de sistema, juntamente com as definições preliminares necessárias para a apresentação deste trabalho. Para contextualizar a metodologia para a detecção de falhas, a Seção 5.2 detalha a arquitetura e as características de funcionamento da NFV-FD. Os experimentos e as conclusões do trabalho são apresentados nas Seções 5.3 e 5.4, respectivamente.

5.1 Modelo de Sistema

Neste trabalho considera-se um sistema distribuído assíncrono de topologia arbitrária composto por um conjunto Π de n processos/*hosts*, incrementado com detector de falhas [Chandra e Toueg, 1996]. Tanto os processos como os enlaces falham por colapso (*crash*), ou seja, deixam de executar suas tarefas prematuramente. O monitoramento dos processos é reali-

zado por um detector de falhas que pode cometer enganos. O monitoramento dos processos, pelo detector de falhas, resulta nos seguintes estados: suspeito (*S: suspect*), não suspeito (*T: trusted*) ou inatingível (*U: unreachable*). O estado *S* é definido quando uma resposta de uma requisição de vida não é obtida dentro de um intervalo específico de tempo. Entretanto, se a mensagem chegar dentro do intervalo de tempo, o respectivo processo é considerado no estado *T*. O estado *U* ocorre quando é reportada falha em canais de comunicação que, consequentemente, deixam processos inacessíveis.

Assume-se que falhas não afetam a comunicação entre o detector de falhas e o controlador da rede, em outras palavras, não há perda de mensagens. Além disso, o canal não cria, não duplica e não altera mensagens. A rede é também formada por um controlador que nunca falha.

5.2 Proposta de uma NFV para Detecção de Falhas

Nesta seção é apresentado o funcionamento proposto de detectores de falhas como uma NFV (Seção 5.2.1). A arquitetura em que a NFV-FD está sendo proposta é ilustrada na Seção 5.2.2, por fim, são apresentados os mecanismos para detecção de falhas nos processos (Seção 5.2.3) e enlaces de comunicação (Seção 5.2.4).

5.2.1 Detectores de Falhas como uma NFV

Ao longo dos anos, diversas propostas de algoritmos de detecção de falhas surgiram. Consequentemente, suas funcionalidades foram sendo exploradas em vários aspectos. Entre as suas vantagens pode-se destacar a sua utilização como serviço independente da aplicação [Felber et al., 1998]. A utilização dos detectores de falhas como serviços vão desde sua implementação como *middleware* [Zia et al., 2009], até serviços mais próximo do ambiente de rede como a implementação via protocolo SNMP apresentado no Capítulo 4 [Turchetti e Duarte Jr., 2015a].

Neste contexto, a proposta do presente trabalho é dispor o serviço para detecção de falhas via uma função de rede virtualizada. A virtualização do serviço para detecção de falhas é implementada em uma rede OpenFlow [Openflow, 2015], onde toda a lógica para a troca de informações dos dispositivos na rede é realizada por regras instaladas por um controlador. Para realizar a detecção dos estados dos processos neste tipo de rede, a NFV-FD se comunica com o controlador e extrai informações relevantes para o processo de monitoramento.

Neste sentido, o primeiro desafio da NFV proposta é habilitar a comunicação do serviço de detecção de falhas com o controlador OpenFlow. Outra tarefa importante é a criação de regras que permitem especificar os tipos de pacotes que devem ser encaminhados para o processamento da NFV-FD. Estas informações irão auxiliar na obtenção dos atributos dos processos a serem monitorados, bem como permitir determinar o próprio estado destes processos. Além disso, são também obtidas informações dos estados dos enlaces de comunicação a partir do monitoramento realizado pelos próprios dispositivos da rede. A integração da NFV-FD com o controlador OpenFlow é apresentada na próxima seção.

5.2.2 NFV-FD: Arquitetura

A arquitetura ilustrada na Figura 5.1 apresenta a integração do mecanismo para monitoramento dos processos proposto neste trabalho. O controlador OpenFlow é composto por módulos (na Figura 5.1, *Firewall*, *Hub*, ...) que possibilitam a integração de novas funcionalidades ao ambiente em execução, proporcionando acesso direto ao controlador da rede. O

controlador separa as funções pertinentes ao controle dos dispositivos na rede OpenFlow, como a criação de regras de comunicação, gerenciamento de topologia e de dispositivos, interface para acesso via Web, entre outras. O controlador possui ainda uma API para a interface **REST**[1], facilitando a configuração em rotinas internas aos módulos que estendem as funcionalidades da arquitetura.

Inicialmente, para a integração do controlador com a NFV-FD, um módulo denominado de FDMod é definido. O FDMod trabalha como um filtro de pacotes analisando informações do cabeçalho. Sua função é auxiliar na obtenção dos atributos dos processos que serão monitorados pela NFV-FD. A obtenção destes atributos é realizada com base em regras pré-definidas. As informações são extraídas do cabeçalho do pacote; um exemplo de atributos é o endereço IP e porta do processo a ser monitorado.

A NFV-FD, sempre que recebe informações de atributos pertencentes a um novo processo, inicia o mecanismo de monitoramento do respectivo processo. O recebimento das informações dos atributos dos processos é feito com base em regras que podem ser instaladas por uma aplicação de rede[2]. Uma regra deve descrever as características de comunicação da aplicação distribuída. Regras podem ser criadas de acordo com o protocolo e porta de comunicação da aplicação distribuída, como exemplo, considere as informações de um pacote capturado pelo FDMod:

[in_port=1,dl_dst=01:00:5e:00:00:01,dl_src=00:00:00:00:00:01,nw_dst= 230.0.0.1, nw_src=10.0.0.1,nw_proto=17,nw_tos=0,tp_dst=4446,tp_src=4446]

Uma regra é criada com base nos seguintes campos: o endereço de destino (*nw_dst= 230.0.0.1*), o protocolo UDP (*nw_proto=17*) e a porta da camada de transporte (*tp_dst=4446*). O FDMod encaminha o pacote para a NFV-FD executar sua tarefa, por exemplo iniciar o monitoramento do seguinte processo: *nw_src=10.0.0.1* porta *tp_dst=4446*.

Em outras palavras, todas as mensagens recebidas pela **API OpenFlow** (ilustrada na arquitetura) são repassadas ao FDMod. O FDMod filtra estes pacotes para repassar à NFV-FD somente os pacotes que se encaixarem nas regras, ou ainda, mensagens de eventos notificados pelos *switches* que são descritos na seção 5.2.4.

5.2.3 Mecanismo para Detecção de Falhas em Processos

O monitoramento dos processos realizado pela NFV-FD ocorre através de requisição de mensagens de monitoramento (*liveness request*), que são periodicamente enviadas aos processos monitorados. Isto é, utiliza-se o modelo *Pull* descrito na Seção 3.1.1. Considerando aspectos para o cômputo do *timeout*, nós utilizamos a mesma estratégia apresentada na Seção 4.2, ou seja, estima-se o tempo de chegada da próxima mensagem (EA) acrescida de uma margem de segurança (α), como descreve a Expressão 4.2.

5.2.4 Mecanismo de Detecção de Falhas em Enlaces de Comunicação

Entre as diversas vantagens em utilizar a NFV-FD em uma rede OpenFlow, pode-se destacar a possibilidade de detectar falhas em enlaces de comunicação. Para executar esta função, a NFV-FD faz uso de um mecanismo de descoberta de topologia que é executado pelos próprios *switches* OpenFlow.

O mecanismo de descoberta de topologia é realizado por um protocolo denominado de *Link Layer Discovery Protocol* (LLDP) [IEE, 2009]. O protocolo OpenFlow define este

[1] *REpresentational State Transfer*

[2] Aplicação que utiliza a interface REST para configurar as regras.

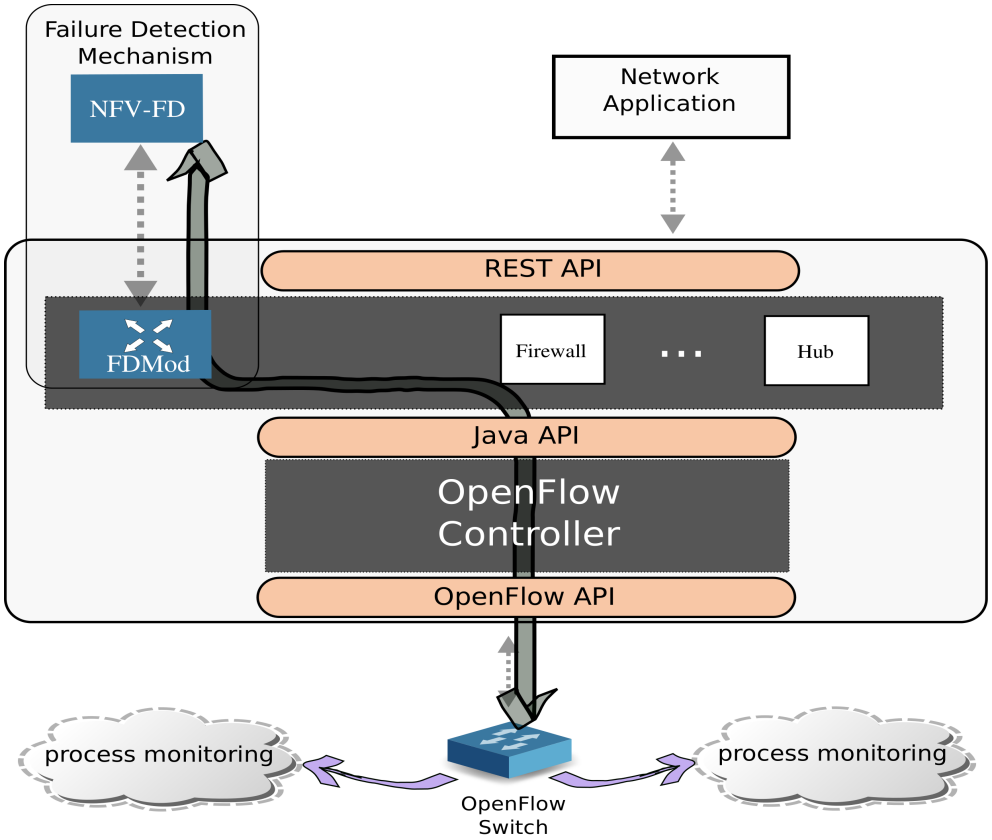


Figura 5.1: Integração da NFV-FD na arquitetura do controlador OpenFlow.

mecanismo de descoberta através de pacotes do tipo *packet in* e *packet out*. Pacotes do tipo *packet out* são transmitidos entre *switches* OpenFlow periodicamente através do protocolo LLDP. Quando um *switch* OpenFlow recebe um pacote *packet out*, pacotes LLDP são transmitidos para todas as portas de saída. Quando o correspondente *switch* OpenFlow recebe o pacote LLDP, ele envia um *packet in* para o controlador comunicando o *status* de um enlace [Sharma et al., 2011].

Existem três situações em que o controlador detém informações sobre o *status* de um enlace [Openflow, 2015]:

- *Link up (Switch → Controller)*: Quando um *switch* OpenFlow detecta um novo *switch*, ele encaminha uma mensagem para o controlador comunicando a nova conexão.
- *Link down (Switch → Controller)*: Quando um *switch* OpenFlow fica sem receber mensagens LLDP de outro *switch* por um período de tempo (*timeout*), ele encaminha uma nova mensagem para o controlador comunicando a perda do enlace.
- *Get Links (Controller → Switch)*: O controlador pergunta para o *switch* sobre o estado dos enlaces, é mais comum ocorrer esta situação quando um controlador se conecta pela primeira vez ao *switch*.

Para repassar os eventos detectados pelos *switches* para a NFV-FD, o FDMod tem acesso às informações da topologia da rede. O FDMod busca estas informações através do controlador OpenFlow. O controlador recebe informações dos eventos listados anteriormente, através de uma função denominada de *Link Discovery*. Por exemplo, quando ocorrem eventos do tipo *Link up* ou *Link down* o controlador recebe a mensagem e repassa para o FDMod.

O FDMod, por sua vez, repassa informações sobre o evento ocorrido para a NFV-FD. Por exemplo, *'port s2-eth2 changed: DOWN'* indica que a porta 2 do *switch* 2 está sem conexão.

Se houver processos que estão sendo monitorados nesta porta, do referido *switch*, a NFV-FD rotula o estado do processo como *U*, indicando que o processo está inacessível.

Vale ressaltar que a utilização de uma rede SDN possibilita a integração de protocolos ou dispositivos que, em geral, necessitam de interfaces específicas de comunicação. Exemplo disso é a comunicação com o protocolo LLDP. Neste contexto, acredita-se que outros benefícios podem ser explorados como a utilização de outras estratégias de detecção de falhas em enlaces, mais efetivas. Um exemplo é o protocolo *Bidirectional Forwarding Detection* (BFD) que foi estendido para implementar *fast recovery* em redes SDN [van Adrichem et al., 2014].

5.3 Avaliação da NFV-FD

Nesta seção são apresentados resultados experimentais e diferentes estratégias para a implementação da NFV-FD. Inicialmente, a função virtualizada é implementada junto ao controlador SDN e, posteriormente, em uma entidade externa ao controlador. Além disso, os experimentos avaliam o desempenho da NFV-FD usando máquinas virtuais tradicionais e através do uso de *containers*. Avalia-se também a qualidade do serviço provido pela NFV-FD a partir do ponto de vista da aplicação que usa um algoritmo para difusão confiável. Este algoritmo é descrito na primeira subseção.

O ambiente para execução dos experimentos é baseado em uma rede OpenFlow implementado com o controlador Floodlight[3]. Os *containers* são executados na plataforma Docker[4], ao passo que as máquinas virtuais utilizam como hipervisor o Virtualbox[5]. Os experimentos são executados em um computador com processador Intel Core i5 CPU 2.50GHz, com 4 núcleos e sistema operacional Mint 17, com kernel 3.13.0-24. A rede foi criada e virtualizada através da ferramenta Mininet[6].

5.3.1 Algoritmo de Difusão Confiável

Uma definição formal de difusão confiável foi apresentada no Capítulo 3. Relembre que o algoritmo faz uso das seguintes primitivas: *broadcast(m)* e *deliver(m)*, onde *m* é uma mensagem. A primitiva *broadcast(m)* significa que *m* será transmitida para todos os processos pertencentes ao conjunto Π de processos (ver Seção 5.1), a primitiva *deliver(m)* significa que a mensagem *m* pode ser entregue a este conjunto de processos. Estas primitivas são apresentadas no Algoritmo 2 e os detalhes de seu funcionamento são descritos na sequência.

Note que diferentemente dos conceitos apresentados na Seção 3.3, o Algoritmo 2 faz uso de um detector de falhas, e funciona como explicado na sequência.

Primeiramente, o algoritmo executa o evento *_Init* inicializando as seguintes variáveis: *delivered* (registro das mensagens já entregues), *correct* (estados dos processos atualizados pela NFV-FD) e *from*$[p_i]$ (armazena cópia da mensagem original enviada pelo emissor). Quando uma mensagem é transmitida por difusão confiável, o evento *_Broadcast* é invocado. O evento *_Deliver* é executado para entregar uma mensagem *m* transmitida por algum processo do conjunto de processos Π. A mensagem *m* é entregue e o estado do processo emissor (p_i) é verificado. Se $p_i \notin correct$, uma nova difusão confiável é inicializada, sendo encaminhada a mensagem original já gravada em *from*$[p_i]$. Finalmente, o evento *_Crash* é invocado quando um processo

[3]http://www.projectfloodlight.org/
[4]https://www.docker.com/
[5]https://www.virtualbox.org/
[6]http://mininet.org/

Algorithm 2: Algoritmo para difusão confiável [Guerraoui e Rodrigues, 2006].

```
1  upon event ⟨ _Init ⟩ do
2     delivered := ∅;
3     correct := Π;
4     forall p_i ∈ Π do
5        from[p_i] := ∅;

6  upon event ⟨ _Broadcast (m) ⟩ do
7     run ⟨ broadcast (Data, self, m) ⟩

8  upon event ⟨ _Deliver (Data, s_m, m) ⟩ do
9     if (m ∉ delivered) then
10          delivered := delivered ∪ {m}
11          run ⟨ deliver (s_m, m) ⟩;
12          from[p_i] := from[p_i] ∪ {(s_m, m)}
13              /* (Caso 2: envia m) */
14          if (p_i ∉ correct) then
15              run ⟨ broadcast (Data, s_m, m) ⟩;

16 upon event ⟨ _Crash (p_i) ⟩ do
17    correct := correct \ {p_i}
18          /* (Caso 1: envia m) */
19    forall (s_m, m) ∈ from[p_i] do
20          run ⟨ broadcast (Data, s_m, m) ⟩;
```

p_i está suspeito e a mensagem m, transmitida por p_i, ainda não foi entregue. Uma cópia de m é novamente transmitida por difusão confiável para futura entrega.

Em síntese, dois eventos que são dependentes da NFV-FD forçam um processo a retransmitir uma mensagem:

- Caso 1: quando um processo percebe a falha do emissor antes de entregar m.
- Caso 2: quando um processo entrega m e percebe que o emissor falhou.

A seguir, são apresentados os experimentos realizados com a NFV-FD utilizando o algoritmo para difusão confiável ilustrado nesta seção.

5.3.2 Avaliação do Serviço para Detecção de Falhas

No primeiro experimento são reportados resultados da implementação do detector de falhas implementado junto ao controlador SDN em comparação à implementação em uma entidade externa ao controlador. Para este experimento a rede é virtualizada com uma topologia simples utilizando 1 *switch*, 1 controlador remoto e 4 *hosts* executando o algoritmo de difusão confiável. A NFV-FD é configurada para monitorar os *hosts* com periodicidade de 1000ms. Os *hosts* que executam a difusão confiável são utilizados para gerar fluxo na rede e, portanto, também configurados para transmitirem mensagens a cada 1000ms.

O objetivo deste experimento é avaliar o impacto que a NFV causa quando hospedada no controlador. Alguns autores, incluindo Batalle [Batalle et al., 2013], mencionam que uma implementação integrada ao controlador traz benefícios, por exemplo, a aplicação pode ter uma visão completa da rede. Para este experimento é avaliado o desempenho quanto à utilização de CPU levando em consideração: (1) o desempenho do controlador executando sem a presença da NFV, (2) o desempenho da NFV-FD como uma entidade separada e, (3) o desempenho de ambos (NFV-FD e controlador) executando na mesma CPU. Cada experimento foi executado no intervalo de 60 min. Os resultados apresentados na Figura 5.2 são dados médios de 5 execuções, sendo também apresentados valores mínimos e máximos observados.

Observe na Figura 5.2 que quando a NFV-FD é executada com o controlador (curva '*NFV-FD and Controller'*) a utilização média de CPU é de 4,49%. Ao passo que quando o controlador é executado separadamente, a utilização de CPU é de aproximadamente 2,67% (curva '*Controller'*). Este valor reflete basicamente o processamento para o gerenciamento dos pacotes transmitidos por difusão confiável pelos 4 *hosts*. O custo de execução da NFV-FD separada é apresentado pela curva 'NFV-FD' e demonstrou uma utilização média de 2%. Assim, pode-se concluir que colocando a NFV-FD no controlador, simplesmente aumenta-se a necessidade de recursos requeridos pelo controlador SDN. Dessa maneira, é possível concluir que esta estratégia não é escalável: há claramente um limite no número de funções que podem ser implementadas nesta abordagem. Além disso, conforme o número de funções cresce ou a rede aumenta, o impacto no desempenho global do ambiente, será certamente maior.

Para os experimentos da Figura 5.3, a rede foi virtualizada com uma topologia contendo 3 *switches*, 1 controlador remoto e 4 *hosts* executando o algoritmo de difusão confiável.

A Figura 5.3 apresenta o tempo de detecção (T_D) considerando falhas ocorridas em *hosts* e em enlaces de comunicação. Para todos os casos, o tempo apresentado para detecção são valores médios, onde 10 falhas são forçadas no ambiente. A falha é implementada pelo bloqueio das mensagens de monitoramento[7] forçando o estouro do *timeout*. O pior caso (*Worst-Case*)

[7]Mensagens do protocolo LLDP (falha em enlace) e por mensagens da NFV-FD (falha em *host*).

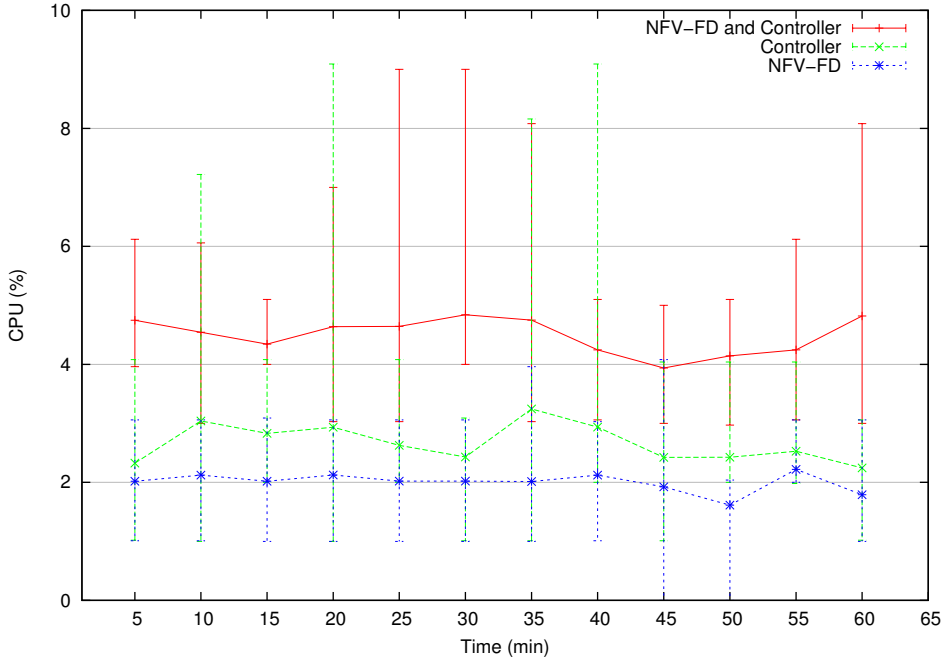


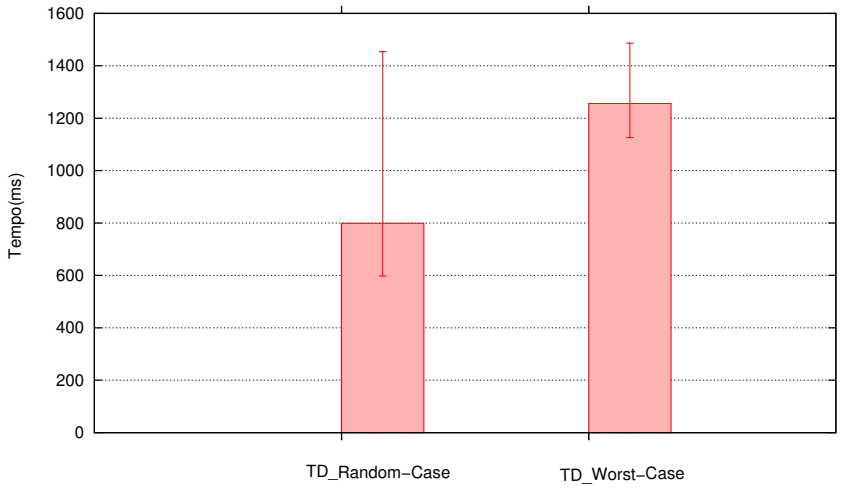
Figura 5.2: Utilização de CPU: desempenho da NFV-FD executada dentro e fora do controlador SDN.

para o T_D é computado considerando-se que a falha ocorre concomitantemente com a resposta de uma requisição de monitoramento.

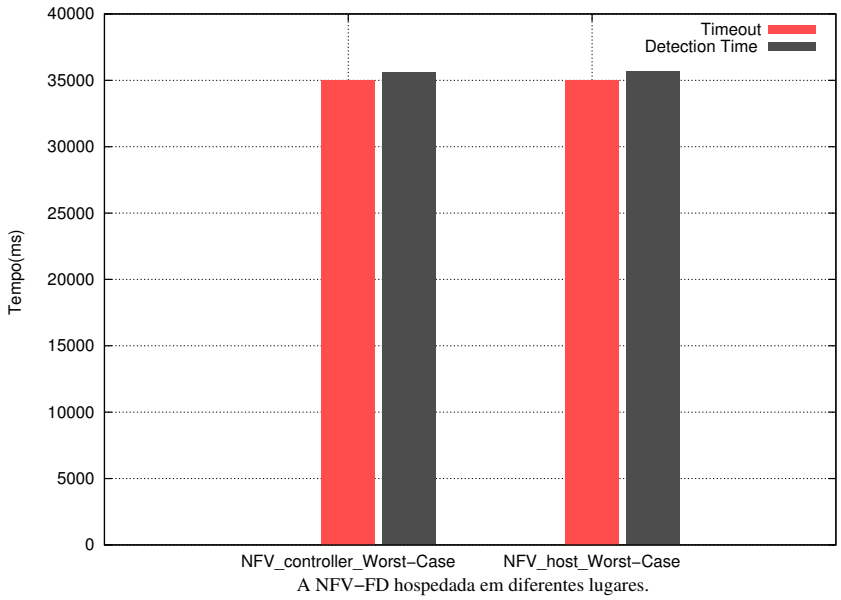
Para o gráfico da figura 5.3(a) o tempo de detecção é computado no início da falha de um *host* até todos os demais *hosts* da rede (3 *hosts*) serem notificados da falha. A notificação é verificada somente quando o algoritmo de difusão confiável é informado. Considerando o pior caso, o tempo de detecção médio é de 1256ms, enquanto que o máximo observado é de 1454,23ms. Tendo em vista que o *timeout* possui um cômputo adaptativo, o valor máximo de T_D pode ser causado por atrasos nas mensagens de *liveness request* ou pelo próprio processamento necessário para executar o algoritmo de difusão confiável.

Para o caso em que as falhas ocorrem em instantes aleatórios (*Random-Case*), é possível observar que há uma grande variação no T_D, onde os valores médios se concentram aproximadamente em 800ms. O valor mínimo observado neste experimento é de 598ms. Este melhor resultado ocorreu, provavelmente, no caso em que a falha foi executada instantes antes do Δ_{to} expirar, aliado a isso, ocorreu a notificação dos demais *hosts*. Considerando a detecção e notificação de uma falha por um único *host*, o melhor tempo observado pelo algoritmo de difusão confiável foi de 22.68ms. Ressalta-se que este experimento foi beneficiado pela utilização de uma rede SDN, pois a implementação deste tipo de cenário em um ambiente não virtualizado, não é uma tarefa trivial, uma vez que exige-se uma forte sincronização dos relógios.

Para o gráfico da figura 5.3(b), o objetivo é avaliar o tempo de detecção de uma falha no enlace de comunicação. Este evento ocorre através da notificação realizada pelo protocolo LLDP para a NFV-FD. Neste sentido, avalia-se o tempo de detecção com a NFV-FD em dois locais: no controlador e no *host*. Pode-se observar que o tempo de detecção do gráfico 5.3(b) é elevado se comparado ao gráfico da figura 5.3(a). Isso se deve ao fato de que manteve-se o valor padrão para a variável `LINK_TIMEOUT` sendo de 35 segundos no controlador *Floodlight*. Verificamos que o tempo de detecção é ligeiramente menor quando a NFV-FD está junto ao controlador, se comparado com a NFV-FD localizada no *host*. A diferença média observada foi de 120ms, o que corresponde ao tempo de comunicação entre o controlador e o *host*. Por outro lado, vimos no



(a) Falha no *host*: TD computado até todos os *hosts* da difusão confiável serem notificados.



(b) Falha no enlace: TD computado até a NFV-FD ser notificada.

Figura 5.3: Análise da qualidade do serviço para o tempo de detecção.

experimento anterior que este tempo superior no período de detecção pode ser compensado com a redução no uso dos recursos computacionais do controlador.

Ressalta-se que o experimento para a detecção de falhas em enlaces reportado pelo protocolo LLDP, foi avaliado para demonstrar a sua viabilidade e os possíveis benefícios no uso de redes SDN. Uma redução no tempo de detecção pode ser obtida diminuindo o valor padrão do `LINK_TIMEOUT`.

5.3.3 Implementação da NFV-FD com Containers: Resultados Experimentais

Neste experimento são avaliados os benefícios da implementação da função virtualizada de rede NFV-FD em *containers*. Virtualização baseada em *containers* é uma alternativa aos tradicionais hipervisores. *Hipervisores* criam uma camada de virtualização que executa como

Tabela 5.1: Container versus VM: tempo de instaciação.

Sistema de Virtualização	**Média (s)**	**Máx (s)**	**Mín (s)**	**Desvio Padrão (s)**
NFV baseado em container	1,57	1,65	1,53	0,04
NFV baseado em VM	38,54	40,07	29,04	3,35

uma aplicação no nível do Sistema Operacional (SO) [Cziva et al., 2015]. Aplicações baseadas em *containers* executam diretamente no SO e não empregam a virtualização do hardware. Portanto, *containers* devem ser mais eficientes do que as VMs (*Virtual Machines*) e permitem maior densidade de aplicações no mesmo host [Anderson et al., 2016]. Os experimentos nesta seção comparam a NFV-FD executando em VMs no hipervisor Virtualbox e em *containers* na plataforma Docker.

Para executar a NFV-FD proposta em *containers*, utilizou-se a arquitetura descrita na Figura 5.4. Esta arquitetura mostra VMs que trocam informações através do *switch* OpenFlow (OVS: Open vSwitch[8]). O fluxo de dados é redirecionado para a NFV-FD conforme apresentado na Figura 5.1 usando o FDMod (vide Seção 5.2.2), que é o módulo responsável por filtrar os pacotes e encaminhá-los às NFVs de acordo com regras pré-definidas.

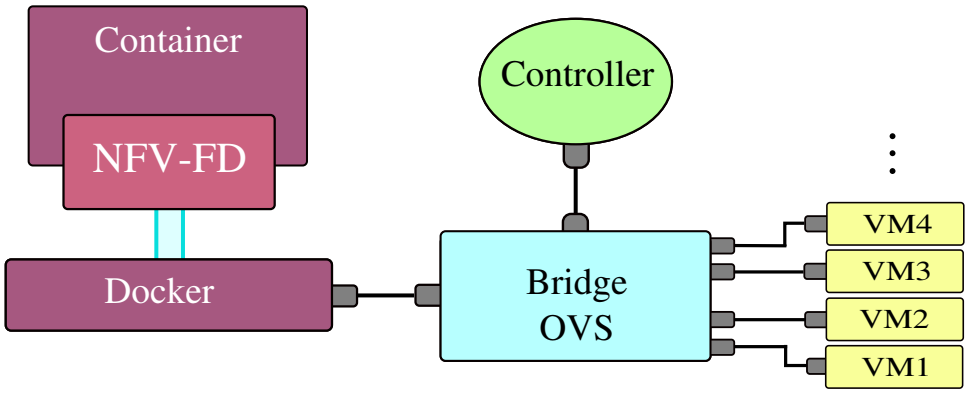


Figura 5.4: NFV-FD executando em *containers*.

Para esta primeira série de experimentos, é medido o tempo para instanciar cada sistema, isto é, mede-se o tempo que um *container* ou uma VM leva para inicializar e parar sua execução. Um *script* incluído no arquivo 'rc.local' é usado para desligar o sistema. Os resultados apresentados na Tabela 5.1 descrevem os valores da média, máximo, mínimo e do desvio padrão. Para cada sistema de virtualização foram coletadas 10 amostras. Ambos os sistemas de virtualização são preparados para executar a NFV-FD no sistema operacional Linux Ubuntu 14.04.

Observe na Tabela 5.1 que quando a NFV é executada em um *container*, tem-se os melhores resultados para o tempo de instanciação. Isto é, a NFV executando como um *container* melhora o tempo de instanciação em aproximadamente 95% comparado ao tempo de instanciação de uma VM. Além disso, note que o valor mínimo, máximo e desvio padrão apresentam baixa variação quando executando em *containers*, isso indica que os valores coletados nos experimentos estão próximos da média.

[8]http://openvswitch.org/

A segunda série de experimento foi executada para medir a utilização de CPU e memória. Para este experimento a rede consiste na topologia da Figura 5.4: um switch (OVS), um controlador SDN e quatro hosts executando o algoritmo de difusão confiável. A NFV-FD é configurada para enviar mensagens de monitoramento em uma periodicidade de 1000ms. Os hosts que executam o algoritmo de difusão confiável são utilizados para gerar fluxos de dados na rede. Eles também são configurados para transmitirem mensagens a cada 1000ms.

As Figuras 5.5 e 5.6 mostram os resultados em um período de uma hora de execução. É fácil perceber as vantagens do uso de *containers* – em particular a utilização de CPU, reduzida em aproximadamente 89,3%. Similar resultado pode ser visualizado na Figura 5.6, onde os *containers* também reduzem a utilização de memória em aproximadamente 74,5%. Acreditamos que estes resultados estão de acordo com os benefícios que os *containers* proporcionam, por exemplo, eles não precisam virtualizar o sistema de hardware como: virtualização de processamento (*vCPU*), rede (*vNIC*) e armazenamento (*vStorage*), além disso eles compartilham o *kernel* do sistema hospedeiro. Portanto, conclui-se que *containers* são excelentes para a virtualização de funções de rede, pois utilizam uma tecnologia de virtualização mais leve.

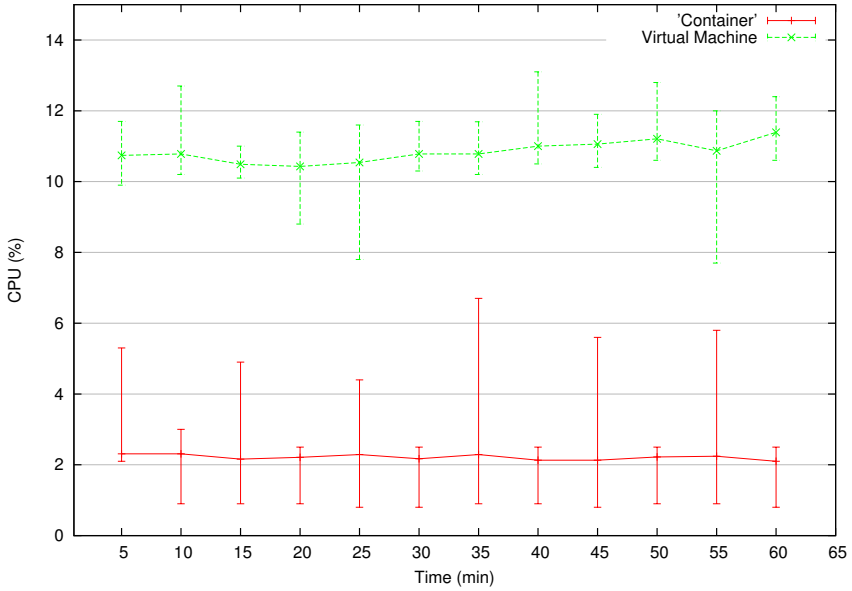


Figura 5.5: Container versus VM: utilização de CPU.

5.4 Considerações Parciais

Neste capítulo foi proposta uma VNF para detecção de falhas de processos e enlaces de comunicação em uma rede SDN. A função é denominada de NFV-FD e utiliza um módulo no controlador SDN que auxilia no monitoramento dos processos. A estratégia proposta não implica em nenhuma modificação no cabeçalho do pacote, protocolo OpenFlow ou *switches* de comunicação. Para avaliar a NFV-FD, um algoritmo para difusão confiável foi implementado e utilizado nos experimentos. O tempo de detecção de falhas e o uso dos recursos computacionais foram avaliados. Os experimentos mostraram que a implementação de uma NFV junto ao controlador causa um impacto relevante no uso do processador sendo, portanto, uma boa estratégia de projeto removê-la do controlador, mesmo com o ligeiro aumento no tempo de detecção de falhas demonstrado nos experimentos. Também foi apresentado um estudo comparativo que possibilita mostrar, entre os sistemas de virtualização analisados, qual é o mais apropriado para

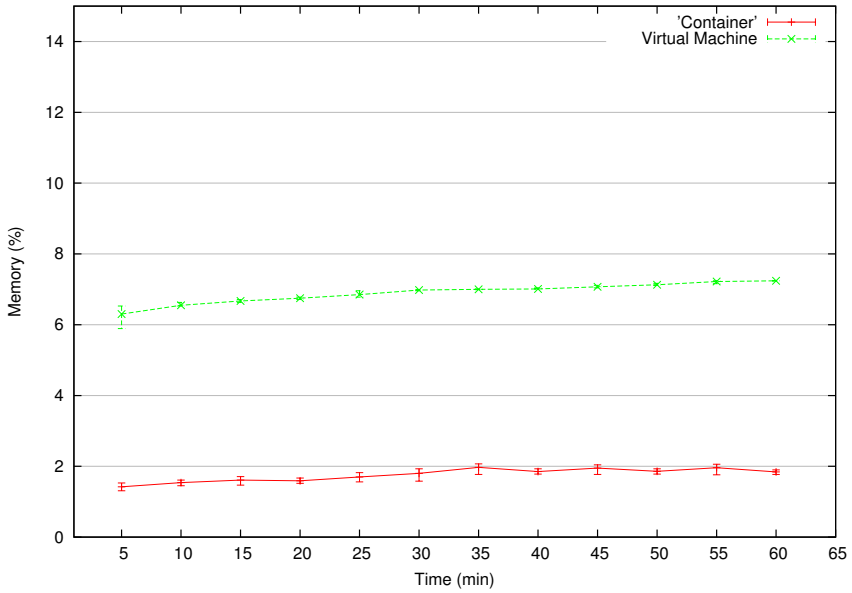


Figura 5.6: Container versus VM: utilização de memória.

a virtualização de uma NFV. O estudo mostrou que *containers* oferecem vantagens quando comparados com hipervisores. As vantagens são em termos de tempo de instanciação, utilização de memória e de CPU. Portanto, concluímos que *container* é uma excelente alternativa para a virtualização de funções de rede.

Capítulo 6

Uma Estratégia para Melhorar a Precisão do Timeout de Detectores de Falhas

Detectores de falhas fazem uso de *timeouts* para identificar a falha de processos monitorados. É muito importante que o *timeout* seja preciso, para não causar falsas suspeitas, nem demorar muito tempo para detectar uma falha que realmente ocorreu. Neste capítulo é proposta uma estratégia denominada $tuning_{\phi}$, que reajusta o valor do *timeout* de acordo com os tempos de comunicação calculados, buscando refletir o comportamento real da rede. Em especial, adaptamos o cálculo proposto por Jacobson [Jacobson, 1988], usado no protocolo TCP, ajustando o peso da constante responsável por multiplicar o valor correspondente aos tempos de comunicação. Dessa maneira, $tuning_{\phi}$ busca tornar o cálculo de Jacobson mais fiel ao comportamento do ambiente. Além disso, a própria estratégia é responsável por indicar pesos para valores que, no algoritmo original, são constantes.

A estratégia proposta neste capítulo é avaliada no contexto de detecção de falhas de processos na Internet, cenário em que a comunicação pode ocorrer com longos períodos de atraso. Mesmo assim, mostramos através dos experimentos executados sobre traces reais que $tuning_{\phi}$ reduz de forma expressiva o número de falsas suspeitas, não atingindo 1% do número de falsas suspeitas cometidas pelo algoritmo original. Além disso, $tuning_{\phi}$ apresenta uma redução no tempo de detecção de falhas mantendo um bom desempenho no tempo médio para correção de falsas suspeitas.

Este capítulo está organizado da seguinte forma. A Seção 6.1 apresenta uma breve descrição sobre o controle do *timeout*, bem como, apresenta argumentos que justificam a estratégia proposta neste capítulo. Na Seção 6.2 é apresentada a estratégia $tuning_{\phi}$ que permite calcular um *timeout* adaptando valores para a constante ϕ. Por fim, os experimentos e as conclusões são descritos na Seção 6.3 e na Seção 6.4, respectivamente.

6.1 Justificativa da Proposta

No contexto das redes de computadores, uma importante tarefa é garantir que os dados enviados por um transmissor foram, de fato, recebidos pelo receptor remoto. Para garantir essa tarefa, o transmissor pode fazer uso de funções da camada de transporte, em especial as oferecidas pelo protocolo TCP (*Transmission Control Protocol*) [Postel, 1981]. Neste contexto, para cada pacote transmitido, o algoritmo implementado pelo protocolo TCP aguarda uma confirmação de retorno enviada pelo receptor remoto. Na ausência de qualquer confirmação, o protocolo TCP utiliza um tempo de retransmissão que permite 'garantir' que os dados não foram recebidos pelo receptor remoto [Paxson et al., 2017]. Esse tempo de duração para a

retransmissão de uma mensagem é denominado de *Retransmission Timeout* (RTO), devendo ser calculado conforme proposto por Jacobson (1988) e sugerido pela RFC (*Request for Comments*) 1122 [Braden, 2017].

O RTO implementado no algoritmo do protocolo TCP é um mecanismo simples, mas utilizado com frequência na computação. O RTO possibilita, entre outras funções, evitar que um processo fique indefinidamente aguardando por uma confirmação. Em geral, o mecanismo de *timeout* é executado por um evento que é escalonado usando um temporizador que expira depois de algum limite de tempo pré-estabelecido [Kebarighotbi e Cassandras, 2011]. O RTO do protocolo TCP é utilizado para calcular o tempo total para uma mensagem percorrer um trajeto compreendido entre um transmissor até o receptor e, consequentemente, o retorno até o transmissor novamente (RTT – *Round Trip Time*). No cálculo do RTO do algoritmo TCP, Jacobson indica pesos para algumas constantes que influenciam diretamente no valor final do *timeout*. Entretanto, alguns autores utilizam além dos pesos padrões para as constantes, outros valores que possibilitam obter resultados diferentes dos originais. Por exemplo, nos trabalhos em [Bertier et al., 2003, Dixit e Casimiro, 2010, de Sá e de Araújo Macêdo, 2010, Moraes e Duarte Jr., 2011] são utilizados outros valores para auxiliarem no cômputo do *timeout* no contexto dos detectores de falhas [Chandra e Toueg, 1996], frequentemente utilizados em sistemas distribuídos tolerantes a falhas.

Segundo Dixit e Casimiro [Dixit e Casimiro, 2010], dependendo dos pesos utilizados para as constantes do algoritmo TCP, o cálculo pode tornar o *timeout* mais ou menos agressivo. Um *timeout* mais agressivo possibilita um menor tempo de detecção de falhas. Por outro lado, um valor menos agressivo pode evitar falsas suspeitas. Em outras palavras, se for considerado que a rede se comporta sem muitas oscilações ou, percebe-se uma redução nos atrasos de comunicação, nestas condições definir um *timeout* grande não trará benefícios, pois o tempo para a detecção de uma falha será maior. Por outro lado, se o comportamento da rede variar bastante ou, percebe-se que os atrasos aumentam linearmente, definir um *timeout* pequeno aumenta as chances do algoritmo de detecção gerar falsas suspeitas. De acordo com a estratégia original proposta por Jacobson e utilizada em diversos trabalhos da literatura, uma vez estabelecidos os pesos das constantes utilizadas para o cômputo do *timeout*, os valores permanecem inalterados ao longo do tempo, mesmo diante de variações no comportamento do sistema.

Com base no exposto, visando melhorar a previsão do *timeout*, propomos uma estratégia para reajustar o valor de uma constante definida e utilizada na fórmula de Jacbson, em busca de valores mais fiéis aos atrasos da rede. O ajuste proposto neste capítulo é baseado em uma análise de tendência que retrata o comportamento de uma série temporal. Uma série temporal é uma sequência de observações coletadas em intervalos determinados ao longo do tempo [Shumway e Stoffer, 2006] que representa, no contexto deste trabalho, os tempos de comunicação entre o transmissor e o receptor. Com a estratégia proposta, os cálculos previstos para o *timeout* seguem mais próximos do comportamento do ambiente de execução. Além disso, a própria estratégia é responsável por indicar um peso para a constante, sem a necessidade de uma indicação prévia.

Portanto, o objetivo é avaliar o comportamento do ambiente em execução, considerando os tempos de comunicação nas trocas de mensagens entre um processo transmissor e um receptor, para utilizá-los em futuras previsões. Em outras palavras, a ideia é avaliar o comportamento atual do ambiente com base em uma análise de séries temporais, para prever a tendência do comportamento futuro. A tendência pode ser calculada com base nos valores do coeficiente angular e do coeficiente linear, obtidos através dos valores da série temporal [Shumway e Stoffer, 2006]. Maiores detalhes sobre esse cálculo são apresentados na próxima seção, onde também explicamos o funcionamento e o algoritmo que implementa a estratégia $tuning_{\phi}$.

6.2 Uma Estratégia para Ajustar o *Timeout* de Acordo com Previsões de Comportamento da Rede

O cálculo do RTO proposto por Jacobson já foi utilizado neste trabalho e foram apresentados os detalhes de seu funcionamento no Capítulo 4. Lembrando que o cálculo proposto por Jacobson é utilizado pelo protocolo TCP para determinar o tempo de duração para a retransmissão de uma mensagem. Este tempo é compreendido como sendo o tempo de transmissão até a confirmação de uma mensagem, também chamado de RTT. No trabalho proposto neste capítulo, a falta de uma resposta de um processo dentro do intervalo de tempo calculado, leva a uma suspeita de falha deste processo. Em especial, o cálculo de Jacobson é utilizado para prever o valor do próximo *timeout* com a finalidade de detectar os estados dos processos, indicando se o processo monitorado está ou não suspeito de ter falhado. O valor do *timeout* é recalculado a cada troca de mensagem de monitoramento.

A seguir, é descrita a estratégia $tuning_{\phi}$ que reajusta o valor do *timeout* adaptando os pesos atribuídos às constantes da fórmula de Jacobson. Além disso, os pesos são adaptados de acordo com os tempos de comunicação calculados durante o procedimento de monitoramento dos processos. Vale ressaltar que as expressões que compõem o cálculo de Jacobson foram detalhadas no Capítulo 4 (vide Expressão 4.3, 4.4, 4.5 e 4.6). Portanto, não serão novamente detalhadas neste capítulo.

6.2.1 A Estratégia $tuning_{\phi}$

No contexto das estratégias implementadas para determinar o valor do intervalo do *timeout*, uma importante característica é configurá-lo de maneira adaptativa. A ideia é buscar funcionalidades que permitam, com precisão, prever o instante de tempo em que a próxima mensagem de monitoramento será recebida pelo processo. Em geral, as abordagens são baseadas em estimativas que visam corrigir o valor do *timeout* de acordo com o comportamento do ambiente.

Neste trabalho é utilizado um algoritmo para detecção de falhas que determina o intervalo de *timeout* conforme atrasos na comunicação. Isto é, o *timeout* representado pela letra α é adaptável a cada mensagem recebida de acordo com a Expressão 4.6 ($\alpha_{(k+1)} = \beta \cdot delay_{(k+1)} + \phi \cdot var_{(k+1)}$) proposta por Jacobson [Jacobson, 1988]. Jacobson indica pesos para constantes, como exemplo ϕ=4. Entretanto, percebe-se que alguns autores utilizam outros valores, como exemplo, em seus experimentos Bertier [Bertier et al., 2003] utiliza ϕ=2. Dixit e Casimiro [Dixit e Casimiro, 2010] propõem utilizar ϕ com valores que podem ser pré-fixados com 1, 2 e 4, sendo que quanto menor o valor, como exemplo $\phi = 1$, transforma o cálculo em um *timeout* mais agressivo, possibilitando um menor tempo de detecção de falhas. Por outro lado, um valor menos agressivo, como $\phi = 4$, ou maior, pode evitar falsas suspeitas.

Além disso, se o comportamento da rede permanece sem oscilações, ou percebe-se uma redução nos atrasos de comunicação, definir um valor maior para ϕ nestas condições não trará benefícios, pois o tempo para a detecção de uma falha será maior. Por outro lado, se o comportamento da rede variar bastante ou, percebe-se que os atrasos aumentam linearmente, definir um valor pequeno para ϕ aumenta as chances do detector de falhas gerar falsas suspeitas.

Com base no exposto, a ideia é propor a estratégia $tuning_{\phi}$ para ajustar o valor de ϕ de acordo com os tempos de comunicação percebidos através das ações de monitoramento, variando o valor entre ϕ_{min} e ϕ_{max}. O ajuste é baseado em uma análise de tendência que retrata o comportamento de uma série temporal representada pelos tempos de comunicação. Lembrando

que uma série temporal é uma sequência de observações coletadas em intervalos determinados ao longo do tempo.

O objetivo é avaliar o comportamento nos tempos de comunicação da troca de mensagens durante o monitoramento dos processos, para utilizá-lo em futuras previsões. Em outras palavras, a ideia é avaliar o comportamento atual do ambiente com base em uma análise de séries temporais, para prever a tendência do comportamento futuro. A tendência pode ser calculada com base nos valores do coeficiente angular (se positivo indica uma tendência crescente, se negativo indica tendência decrescente) e do coeficiente linear obtidos através dos cálculos que consideram os valores da série temporal. Então, a equação de tendência é calculada conforme expressão apresentada a seguir:

$$T = \ell + \partial.t \tag{6.1}$$

Na Expressão (6.1), T é o valor da tendência, ∂ representa o coeficiente angular, ℓ é o coeficiente linear e t é o valor do tempo que representa o período na amostra a ser calculado.

A seguir é descrita a equação que permite obter o valor do coeficiente linear apresentado pela Expressão (6.2) e para o cômputo do coeficiente angular apresentado na Expressão (6.3):

$$\ell = \frac{n.\sum_{i=1}^{n}(t_i.y_i) - \sum_{i=1}^{n} t_i.\sum_{i=1}^{n} y_i}{n.\sum_{i=1}^{n}(t_i^2) - (\sum_{i=1}^{n} t_i)^2} \tag{6.2}$$

$$\partial = \frac{\sum_{i=1}^{n} y_i - \ell.\sum_{i=1}^{n} t_i}{n} \tag{6.3}$$

n é o número de elementos que corresponde ao tamanho da série temporal, y_i representa cada elemento registrado na série temporal, como por exemplo a duração para a resposta de uma mensagem de monitoramento, e t_i é o período que está associado ao elemento y_i, que nada mais é do que a sequência das mensagens registradas na série. Com os valores de ℓ e ∂ obtidos é possível então aplicar os valores na equação de tendência apresentada na Expressão (6.1), para prever o próximo valor de y_i conforme o seu respectivo período (t_i).

Portanto, utilizando o cálculo de tendência busca-se prever o próximo período da série e, consequentemente, calcular o valor apropriado para ϕ. Em outras palavras, basta definir o valor mínimo e máximo (ϕ_{min} e ϕ_{max}) e a própria estratégia $tuning_{\phi}$ é responsável por indicar um peso para a constante ϕ, tornando a constante com valores que são adaptados de acordo com as previsões obtidas através da Expressão (6.1). Além disso, $tuning_{\phi}$ isenta a responsabilidade da indicação de um valor prévio e fixo para a constante ϕ.

Para ilustrar o funcionamento da estratégia $tuning_{\phi}$, considere o Algoritmo 3 onde são apresentadas, resumidamente, as funcionalidades de um detector de falhas que realiza o monitoramento dos processos utilizando a estratégia $tuning_{\phi}$ para calcular o valor do próximo *timeout*. De acordo com o algoritmo, há um processo monitor denominado de p_j e um processo monitorado chamado de p_i. Inicialmente, todo processo monitorado p_i (linha 1) está incluso na lista de suspeitos (*suspect*) e nenhum processo correto é conhecido (linhas 3 e 4).

Periodicamente, p_j executa o evento *_Send* para enviar uma mensagem de requisição de vida ao processo p_i, aguardando por um intervalo de tempo α. Se p_i responder a requisição de vida, p_j atualiza o estado de p_i e, com base em uma análise de tendência, p_j tenta prever o valor do próximo instante de tempo (linha 17). Com o valor previsto para o tempo de chegada da

Algorithm 3: Algoritmo que implementa a estratégia $tuning_{\phi}$.

```
1   Every process p_i ∈ Π:
2   upon event ⟨ _Init ⟩ do
3     suspect := Π;
4     correct := ∅;
5     φ_min := 1;                          ▷ φ_min e φ_max são definidos a priori
6     φ_max := 4;
7     {∃ φ ∈ ℤ+ : φ_min ≤ φ ≤ φ_max};      ▷ indica que φ={1, 2, 3, 4}
8   upon event ⟨ _Send | p_j, type, p_i⟩ do
9     run ⟨ send | self, ARE YOU ALIVE, p_i⟩;   ▷ m{src, type, dst}
10    correct := correct - {p_i};
11    wait for α
12    if (p_i ∉ correct) then
13        suspect := suspect ∪ {p_i};
14  upon event ⟨ _Receive | p_i, YES I AM, p_j⟩ do
15  correct := correct ∪ {p_i};
16  suspect := suspect - {p_i};
17  T:= ℓ + ∂ .t;                  ▷ calcula a tendência para o próximo período
18  φ := ⌈|((T + var) − delay)/var|⌉;   ▷ resultado do arredondamento para cima
19  if (φ > φ_max) then
20      φ := φ_max;
21  α := delay + φ * var;
```

próxima mensagem, calcula-se o valor de ϕ que deve pertencer aos naturais positivos e possuir um valor entre ϕ_{min} e ϕ_{max} indicados nas linhas 5 e 6 respectivamente. Vale ressaltar que a decisão por utilizar os valores apresentados nas linhas 5 e 6 do algoritmo são com base na literatura estudada [Jacobson, 1988, Bertier et al., 2003, Moraes e Duarte Jr., 2011].

Na linha 18 é determinado o valor de ϕ, que é obtido considerando a seguinte condição:

$$\alpha \geq T + var \tag{6.4}$$

onde var é um valor obtido da Expressão (4.5). Além disso, substitui-se α na Expressão (6.4) pela Expressão (4.6) e isola-se a constante ϕ, obtendo-se assim o peso para ϕ de acordo com o valor previsto para o próximo período.

Por fim, utiliza-se o peso encontrado para ϕ na expressão apresentada na linha 21, obtendo-se então o valor corrente de α. Note que α, neste caso, representa o próprio valor do intervalo de *timeout*.

A seguir, são apresentados resultados experimentais que demonstram a eficiência da estratégia $tuning_{\phi}$ descrita nesta seção.

6.3 Resultados Experimentais

Para avaliar a estratégia $tuning_{\phi}$ proposta neste trabalho, utilizou-se um trace real disponibilizado em uma base de dados criada por Hayashibara et. al [Hayashibara et al., 2004] e que está publicamente disponível[1]. Esta base armazena informações de tempos de comunicação entre dois computadores localizados em países diferentes, sendo um computador localizado na Suíça (no *Swiss Federal Institute of Technology in Lausanne*) e o outro localizado no Japão (no *Japan Advanced Institute of Science Technology*). A base descreve os tempos de comunicação entre os dois computadores desconsiderando mensagens perdidas.

Vale ressaltar que os experimentos apresentados nesta seção comparam o algoritmo originalmente proposto por Jacobson, isto é, algoritmo que não adapta o valor de ϕ, com o algoritmo da estratégia $tuning_{\phi}$ apresentado na Seção 6.2, ambos executando um serviço para detecção de falhas. O objetivo dos primeiros experimentos apresentados nas Figuras 6.1 e 6.2 é demonstrar como a estratégia $tuning_{\phi}$ adapta o valor de ϕ de acordo com o comportamento percebido no ambiente de execução. Para estes experimentos foram analisados os resultados das 500 primeiras mensagens armazenadas na base de dados, descrita no parágrafo anterior. Na Figura 6.1(a) é possível observar que o RTT possui um comportamento bastante variado. Verifica-se que, quase periodicamente, o RTT atinge um valor aproximado de 1000ms e cai bruscamente para um valor próximo de 100ms. Neste cenário, o *timeout* computado com o valor de ϕ fixo, isto é, $\phi = 4$, possui variações que acompanham as oscilações nos tempos de comunicação. Podendo-se concluir que a função proposta por Jacobson é originalmente adaptativa.

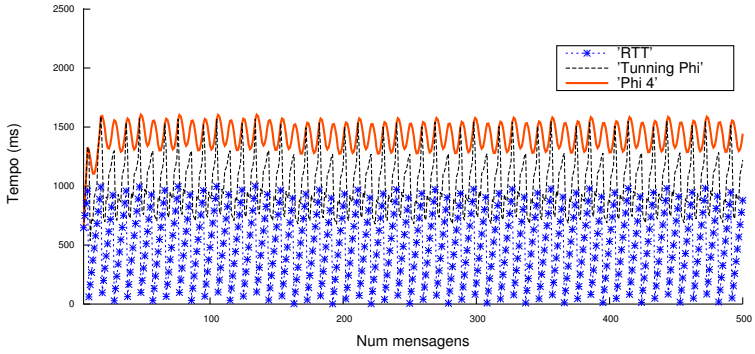
Por outro lado, analisando os valores apresentados pela estratégia $tuning_{\phi}$ é possível perceber que os valores do *timeout* acompanham com mais fidelidade, as oscilações dos tempos de comunicação. A variação do valor de ϕ calculado pela estratégia $tuning_{\phi}$ é mais perceptível na Figura 6.1(b). Neste gráfico, o comportamento quase que periódico do RTT torna a troca de pesos para a adaptação do ϕ praticamente periódica também. Por fim, é possível observar que a troca do peso da variável ϕ, realizada pela estratégia $tuning_{\phi}$, torna a função mais próxima do comportamento real da rede.

O gráfico da Figura 6.2 mostra os valores dos tempos para a correção de falsas suspeitas. O experimento compara a estratégia $tuning_{\phi}$ com a utilização de ϕ fixo, isto é, $\phi = 1$ e $\phi = 4$. Para simular falsas suspeitas, a cada 30 mensagens recebidas foi duplicado o tempo de comunicação para a próxima mensagem. Conforme mostra o gráfico da Figura 6.2, o valor de ϕ fixo com o peso 4 apresentou 6 falsas suspeitas com um T_M médio de 297,9ms, já com $\phi = 1$ observou-se 145 falsas suspeitas mas com um T_M médio de 161,28ms.

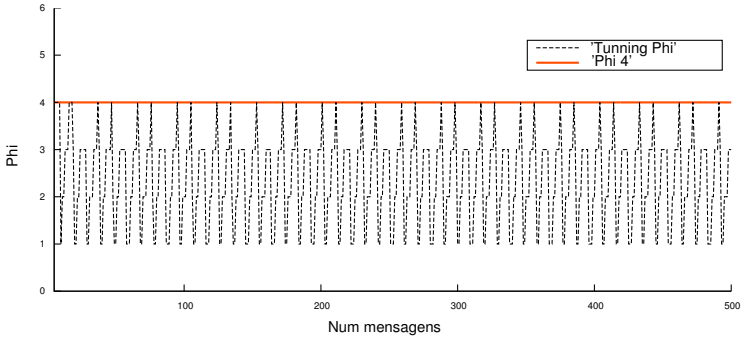
Utilizando a estratégia $tuning_{\phi}$, pode-se observar que ela apresentou valores de T_M para ϕ variando entre 1 a 3, não apresentando nenhuma ocorrência de falsa suspeita com o valor de $\phi = 4$. Entretanto, vale ressaltar que a estratégia $tuning_{\phi}$ atribuiu valores para $\phi = 4$, como mostrado na Figura 6.1(b). Além disso, foram cometidas 14 falsas suspeitas, com um T_M médio de 274,07ms, levemente inferior ao apresentado por ϕ fixo com valor de 4.

É possível observar que com o cômputo do ϕ fixado em 1 diminui-se o tempo médio para correção de falsas suspeitas, apresentando o menor valor obtido, $T_M = 0,38$ms. Entretanto, aumenta-se consideravelmente o número de falsas suspeitas e apresenta também o maior T_M computado, atingindo a máxima de 1168,1ms. A estratégia $tuning_{\phi}$ consegue reduzir o número de falsas suspeitas se comparada ao $\phi = 1$, mas não foi melhor do que ao $\phi = 4$, embora o T_M tenha sido ligeiramente menor para a estratégia $tuning_{\phi}$.

[1] http://ddsg.jaist.ac.jp/en/jst/data.html



(a) Comparação do *timeout* usando $\phi = 4$ com a estratégia $tuning_{\phi}$.



(b) Peso de ϕ ao longo do tempo.

Figura 6.1: Comparação da estratégia $tuning_{\phi}$ com o ϕ fixo.

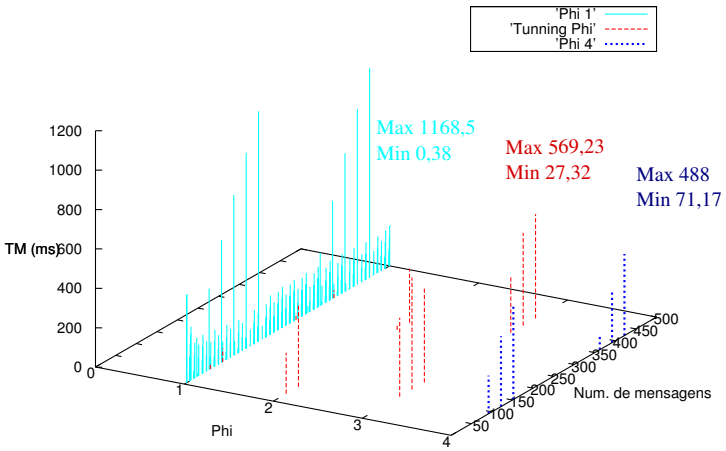


Figura 6.2: T_M para a estratégia $tuning_{\phi}$ e ϕ fixo.

Com base nestes experimentos é possível observar que a estratégia $tuning_{\phi}$ apresentou valores intermediários entre o número de falsas suspeitas e os tempos para as correções de falsas suspeitas. Os resultados apresentados na Tabela 6.1 permitem ver com clareza os resultados para as falsas suspeitas. Para estes resultados foi utilizada toda a base de dados, ou seja, informações coletadas em 7 dias de monitoramento, obtendo um total de 5.822.520 mensagens.

Na Tabela 6.1 é possível observar uma grande vantagem para a estratégia $tuning_{\phi}$ considerando o número de falsas suspeitas cometidas. Vale ressaltar que as falsas suspeitas não foram forçadas como no caso do experimento da Figura 6.2. Aqui, as falsas suspeitas ocorrem quando o cálculo do próximo *timeout* é menor do que o tempo para a chegada da próxima mensagem. É possível observar, para todos os casos apresentados na Tabela 6.1, que a estratégia

Tabela 6.1: Comparação do número de falsas suspeitas cometidas.

(a) ϕ fixo

ϕ	**N. msg**	**N. falsas suspeitas**
1	5822520	1805372
2	5822520	1777
3	5822520	58
4	5822520	47

(b) $tuning_{\phi}$

ϕ	**N. msg**	**N. falsas suspeitas**
1	1408650	38
2	1484547	25
3	2529180	3
4	400143	38

$tuning_{\phi}$ é muito superior quando comparada com o valor de ϕ fixo. De fato, foram cometidas um total de 1.807.254 falsas suspeitas utilizando ϕ fixo, ao passo que, utilizando a estratégia $tuning_{\phi}$ foram observadas 104 falsas suspeitas. O que equivale a dizer que o número de falsas suspeitas cometidas pela estratégia $tuning_{\phi}$ foi menor do que 1%, mais precisamente 0,57% comparada ao ϕ fixo com valores de 1, 2, 3 e 4.

Os experimentos apresentados na Figura 6.3 mostram um comparativo considerando o tempo de detecção (T_D) para uma falha. O T_D é computado a cada falsa suspeita cometida pelo detector de falhas. Neste caso, para $\phi = 3$ a estratégia $tuning_{\phi}$ foi relativamente maior, a razão para este valor é que foram detectadas somente 3 falsas suspeitas (ver Tabela 6.1) sendo todas com valores altos, o que não possibilitou reduzir a média do T_D para este experimento.

Entretanto, observa-se que para os demais casos a estratégia $tuning_{\phi}$ apresentou um T_D menor. Para justificar estes valores, considere o seguinte caso: para uma mesma falha detectada, a estratégia $tuning_{\phi}$ pode estar usando um valor de ϕ menor do que utilizando ϕ fixo, quando isso ocorre, o valor do *timeout* é menor, por consequência a detecção da falha será em menor tempo. Já para $\phi = 1$ ocorrem casos em que uma falha computada pelo ϕ fixo não é calculada pela estratégia $tuning_{\phi}$, pois esta pode estar utilizando um valor de *timeout* maior não caracterizando uma suspeita e não computando o T_D. Em linhas gerais, o T_D total para ϕ foi de 5330,74ms e usando $tuning_{\phi}$ foi de 4761,37ms apresentando uma redução de aproximadamente 10%.

Por fim, podemos concluir que a estratégia $tuning_{\phi}$ apresentou um desempenho consistente, reduzindo o número de falsas suspeitas sem aumentar o T_D, e com bom desempenho no tempo médio para correção de falsas suspeitas.

6.4 Considerações Parciais

O *timeout* é um mecanismo que permite a um detector de falhas tomar a decisão sobre uma suspeita, além disso evita que processos fiquem indefinidamente aguardando por uma resposta. Neste contexto, vimos que o protocolo TCP utiliza uma estratégia proposta por Jacobson para calcular o tempo limite de espera para a confirmação de uma mensagem transmitida. A proposta de Jacobson, por se tratar de um cálculo simples e muito funcional, foi utilizada por diversos autores na literatura. Embora a proposta original seja adaptativa, isto é, o valor é ajustado conforme atrasos percebidos nos tempos de comunicação, neste capítulo

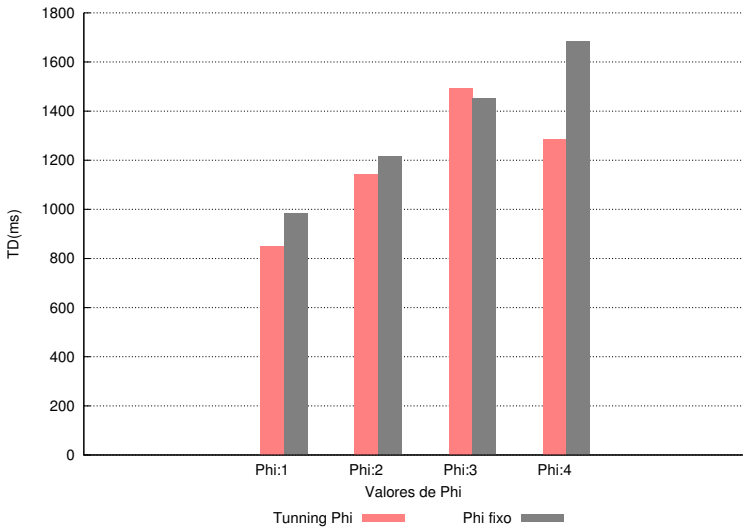


Figura 6.3: T_D para a estratégia $tuning_\phi$ e ϕ fixo.

foi proposta uma estratégia denominada de $tuning_\phi$ que permite, com base na análise de séries temporais, acompanhar com mais fidelidade estes atrasos percebidos na comunicação entre os processos.

Para demonstrar a eficiência da estratégia $tuning_\phi$, seu desempenho foi avaliado utilizando um serviço para detecção de falhas comumente empregado em sistemas distribuídos. Os experimentos realizados foram simulados utilizando uma base de dados que descreve os tempos de comunicação entre dois processos. Considerando os resultados experimentais, $tuning_\phi$ apresentou um excelente desempenho reduzindo de forma expressiva o número de falsas suspeitas, não chegando a atingir 1% do número de falsas suspeitas cometidas pela estratégia original, isto é, quando a constante ϕ foi utilizada com valores fixos. A estratégia $tuning_\phi$ apresentou ainda uma redução no tempo de detecção de falhas mantendo um bom desempenho no tempo médio para correção de falsas suspeitas. Por fim, outra vantagem que vale ser ressaltada é que $tuning_\phi$ isenta a responsabilidade da indicação de um valor prévio e fixo para a constante ϕ.

Capítulo 7

Uma Função Virtualizada de Rede para a Sincronização Consistente do Plano de Controle em Redes SDN

Nas Redes Definidas por Software, o plano de controle é separado do plano de dados. Concretamente, isto significa que a programação dos *switches* SDN é realizada por um controlador. Esse controle é centralizado, portanto tendo restrições em termos de disponibilidade e de escalabilidade. Para resolver esse problema pode-se utilizar estratégias que distribuem o plano de controle. Entretanto, implementar a sincronização entre os múltiplos controladores distribuídos não é uma tarefa trivial. Neste capítulo é proposta uma solução para a sincronização consistente do plano de controle em redes SDN através da implementação de uma função virtualizada de rede denominada de *VNF-Consensus*. *VNF-Consensus* implementa o algoritmo Paxos e com sua utilização os controladores ficam desacoplados da função de manutenção da consistência e podem executar em paralelo suas atividades no plano de controle. Cada controlador SDN possui acesso a uma instância da *VNF-Consensus* através da qual pode receber decisões e enviar dados para serem sincronizados. Dessa maneira, todas as decisões executadas pela *VNF-Consensus* são sistematicamente executadas sem a atuação direta dos controladores.

As vantagens dessa abordagem são: (*i*) os controladores ficam desacoplados da função de manutenção da sincronização do plano de controle e podem executar em paralelo suas atividades no plano de controle; (*ii*) o plano de controle é sincronizado sem aumentar a carga de trabalho dos controladores, pois a *VNF-Consensus* é executada como entidade externa aos controladores, não utilizando os mesmos recursos computacionais; (*iii*) como os controladores não participam diretamente nas decisões do Paxos, o consenso consegue garantir, independentemente do número de controladores operacionais, a conclusão de seus serviços; (*iv*) por fim, a solução proposta não exige nenhuma alteração no funcionamento padrão do protocolo *OpenFlow* ou nos *switches* SDN. Resultados experimentais mostram o desempenho da *VNF-Consensus* e seu impacto na utilização dos recursos, como também os benefícios obtidos por executar o serviço sem a participação direta dos controladores SDN.

Este capítulo está organizado da seguinte forma. A seção 7.1 justifica a proposta apresentada neste capítulo através da descrição do problema da sincronização nos controladores SDN. A seção 7.2 apresenta trabalhos relacionados que tratam da sincronização do plano de controle em redes SDN. Na Seção 7.3 é apresentada a proposta focando no aspecto de implementação e no algoritmo de consenso. Os experimentos e as conclusões do trabalho são apresentados nas seções 7.4 e 5.4, respectivamente.

7.1 Justificativa da Proposta

A programação dos *switches* SDN (*Software Defined Networks*) nas Redes Definidas por Software é realizada por um controlador. Esse controle é centralizado, portanto tendo restrições em termos de disponibilidade e de escalabilidade. Para resolver esse problema pode-se utilizar estratégias que distribuem o plano de controle. Entretanto, implementar a sincronização entre os múltiplos controladores distribuídos não é uma tarefa trivial. O presente trabalho propõe uma solução para a sincronização consistente do plano de controle em redes SDN através da implementação de uma função virtualizada de rede denominada de *VNF-Consensus*. *VNF-Consensus* implementa o algoritmo Paxos e com sua utilização os controladores ficam desacoplados da responsabilidade da sincronização consistente do plano de controle e podem executar em paralelo suas atividades no plano de controle. A implementação e os resultados experimentais mostram os diversos benefícios obtidos pelo uso da solução proposta, em especial a otimização no uso dos recursos computacionais e a redução na carga dos controladores.

Redes Definidas por Software permitem melhorar a flexibilidade e facilitar o gerenciamento das redes de computadores. A estratégia adotada pelas arquiteturas SDN é separar o plano de controle do plano de dados. O controle é, em geral, centralizado e responsável por tomar decisões que ocorrem no plano de dados. Nesta abordagem centralizada, o estado da rede é determinado por um controlador [Ho et al., 2016]. Considerando aspectos de gerenciamento, controle e visão global da topologia da rede, esta abordagem centralizada é, sem dúvidas, atraente. Por outro lado, a implementação do plano de controle centralizado tem, naturalmente, restrições em termos de disponibilidade, desempenho e de escalabilidade [Canini et al., 2015].

Neste sentido, há um consenso de que o plano de controle precisa ser distribuído [Schiff et al., 2016]. Além disso, para que a falha de um controlador possa ser mascarada é necessário aplicar técnicas de redundância no plano de controle [Koponen et al., 2010, Berde et al., 2014]. Entretanto, a sincronização de múltiplos controladores distribuídos não é uma tarefa trivial. Considere, por exemplo, o problema da instalação consistente de novas regras de encaminhamento de pacotes. Se regras conflitantes forem aplicadas estipulando rotas distintas haverá uma patologia na rede podendo, por exemplo, resultar em *loops* indesejados ou rotas contornando serviços de interesse.

Segundo [Canini et al., 2015] um dos principais problemas em aberto no contexto das redes SDN é justamente a necessidade de tornar o plano de controle robusto, sem interferir no desempenho global das funções oferecidas pela rede e, obviamente, preservar a correta operação do plano de dados. Entretanto, as soluções existentes para a construção de um plano de controle robusto, em geral, deixam a cargo do controlador a função de coordenar as ações para a manutenção da consistência realizadas no plano de controle. Em outras palavras, são hospedados nos próprios controladores serviços que permitem *sincronizar* de maneira consistente as informações gerenciadas pelos vários controladores presentes na rede [Koponen et al., 2010, Canini et al., 2015, Ho et al., 2016]. Por outro lado, há soluções que procuram aliviar a carga dos controladores usando os próprios *switches* para sincronizar o plano de dados [Schiff et al., 2016, Dang et al., 2015]. Entretanto, estas são soluções que, em geral, implicam em alterações do funcionamento padrão, como exemplo, mudanças no protocolo *OpenFlow*.

Portanto, acredita-se que aumentar as múltiplas tarefas que os controladores já exercem na rede, não é a estratégia mais adequada, uma vez que a inclusão dessas funcionalidades extras implica no aumento da carga de trabalho. Além disso, segundo os autores em [Karakus e Durresi, 2017] problemas quanto ao desempenho do plano de controle ainda é um assunto que precisa ser investigado tanto pela comunidade acadêmica quanto pela indústria. Sendo assim, o presente trabalho propõe uma solução para a sincronização do plano de controle

em redes SDN, onde os controladores são coordenados por uma função virtualizada de rede (VNF – *Virtual Network Function*). Conceitualmente, uma VNF é a representação de dispositivos de redes em entidades virtuais que são executadas utilizando tecnologias de virtualização baseadas em software [ETSI, 2015b]. A VNF proposta é denominada de *VNF-Consensus* e implementa o algoritmo de consenso Paxos [Lamport, 1998] no ambiente de rede. Dessa forma, a *VNF-Consensus* consegue manter um plano de controle consistente pois sincroniza as ações entre todos os controladores SDN.

7.2 Sincronização dos Dados em Redes SDN

Nesta seção apresentamos os trabalhos relacionados que tratam da sincronização do plano de controle em redes SDN. A principal diferença desses trabalhos para a solução proposta é que, no presente trabalho, há um esforço em permitir a sincronização consistente no plano de controle através do uso de uma função virtualizada de rede.

Para garantir a consistência nas operações de rede, as ações executadas em diferentes controladores SDN precisam ser sincronizadas. Neste contexto, em [Schiff et al., 2016] os autores propõem um *framework* para a sincronização das informações no plano de dados implementadas dentro (*in-band*) dos *switches*. Segundo os autores, protocolos convencionais que executam suas funções fora dos *switches* possuem um alto custo para coordenação e sincronização das informações. Em contraste, soluções *in-band* permitem resolver problemas de acordo (*agreement*) através da troca de poucas mensagens. A solução é baseada no método 'Compare-And-Set' (CAS) abordagem utilizada para a consistência de memória. Esse método garante atomicidade nas transações evitando cenários inconsistentes. Em outras palavras, para evitar inconsistência nas regras instaladas nos *switches* pelos controladores, o comando *FlowMod* é modificado e usa *flags* para definir quais regras devem ser adicionadas ou removidas. Os autores apresentam uma prova de conceito na qual demonstram a eficiência da solução proposta. O mecanismo apresentado pelos autores é simples e pode ser implementado sem a necessidade de hardware ou protocolos extras. Por outro lado, para que os algoritmos propostos funcionem, há a necessidade de efetuar alterações nas *flags* do protocolo *OpenFlow*.

Em [Dang et al., 2015] os autores propõem mover a lógica do algoritmo de consenso Paxos para ser executado dentro da rede. Em particular, propõem que o algoritmo seja executado nos próprios *switches* SDN. Duas abordagens são propostas: a implementação do algoritmo Paxos na sua versão completa, isto é, sem otimizações, para ser executado nos *switches* SDN; e a execução de uma versão otimizada denominada de NetPaxos que evita a implementação de um coordenador (elemento do Paxos) entre os *switches*. Os autores mostram que mover a lógica do consenso para dentro da rede reduz a complexidade das aplicações, reduz a latência das mensagens da aplicação e aumenta o *throughput* das transações. Entretanto, os *switches* precisam executar e assumir papéis que são atribuídos pelo algoritmo Paxos, essa abordagem dificulta sua execução pois exige mudanças no funcionamento padrão da rede, incluindo a alteração do *firmware* nos *switches* SDN.

Em [Ho et al., 2016] os autores propõem uma solução para a sincronização e consistência das informações que são gerenciadas pelos controladores em uma rede SDN. O objetivo é manter a consistência entre múltiplos controladores através da implementação de um algoritmo de consenso. Uma versão ao algoritmo Paxos denominada de FPC (*Fast Paxos-based Consensus*) é proposta. Segundo os autores, o algoritmo FPC reduz a complexidade se comparado ao algoritmo original proposto por Lamport (1998). Em especial, FPC não possui um líder pré-definido, ao invés disso o algoritmo permite que qualquer processo participante possa se tornar o líder (denominado de *chairman*). Uma vez que todos os processos (controladores) tenham realizado

a atualização, o *chairman* retorna a seu papel de *listener* terminando a rodada do algoritmo. Os autores comparam o FPC com o algoritmo de consenso Raft [Ongaro e Ousterhout, 2014] e concluem que o algoritmo proposto apresenta menor tempo para a execução do consenso.

Onix [Koponen et al., 2010] é uma plataforma que permite a implementação do plano de controle como um sistema distribuído mantendo uma visão global da rede. Onix é um sistema distribuído executado em um *cluster* de servidores físicos. No Onix um controlador armazena informações em uma estrutura de dados denominada de NIB (*Network Information Base*). A NIB representa a parte mais importante da plataforma, pois todo o estado da rede é sincronizado através de leituras e escritas controladas pela base de dados. Em especial, Onix representa um conjunto de APIs que fornecem escalabilidade e confiabilidade por replicar e distribuir as informações sincronizadas entre múltiplas instâncias na rede. Se uma informação for alterada, a mesma será propagada por todas as instâncias de NIBs garantindo a consistência através da coordenação de um algoritmo de consenso (Zookeeper [Hunt et al., 2010]). Dito pelos próprios autores, Onix não é uma inovação em si, pois derivam de diversos trabalhos propostos ao longo da história.

Em [Canini et al., 2015], os autores propõem um modelo formal para a comunicação entre o plano de dados e o plano de controle distribuído de uma rede SDN. Em particular, os autores desejam tratar o problema de inconsistência durante a atualização de regras que são instaladas em um ou mais *switches*. Estas regras são as políticas da rede e a solução deve, informalmente, garantir que um pacote que está trafegando na rede seja processado por exatamente uma única política, mesmo em situações em que ela (a política) eventualmente seja atualizada. Os autores apresentam um protocolo denominado de *ReuseTag* que usa uma abordagem de máquina de estados para implementar a ordem total das atualizações das políticas, a solução assume que o plano de dados é como se fosse uma estrutura de memória compartilhada onde os controladores são os processos que fazem leituras e escritas nessa memória. Dado um limite máximo de latência na rede e assumindo um número máximo de *f* processos/controladores falhos, o protocolo *ReuseTag* funciona corretamente.

O OpenDaylight[1] (ODL) é um controlador de rede SDN que oferece em sua arquitetura diversas funcionalidades, em especial a possibilidade de executar e de sincronizar múltiplos controladores SDN. Essa arquitetura é denominada de ODL *Clustering* e oferece um mecanismo de integração entre múltiplos processos e aplicações. Cada controlador possui seu próprio local de armazenamento e a replicação dos dados ocorre através de caches distribuídas aplicando o esquema Infinispan[2]. Toda a comunicação e notificação entre os controladores no ODL *Clustering* ocorre usando um grupo de ferramentas denominado de *Akka*. Essa comunicação ocorre somente entre os controladores da rede, as decisões e a consistência são garantidas pelo uso do protocolo de consenso Raft. O Raft é utilizado para coordenar as atualizações que são executadas entre os controladores SDN. Note que toda a sincronização das informações exige a execução de algoritmos que estão plugados diretamente em cada controlador. Como resultado há um esforço adicional executado pelos controladores a fim de obter a consistência das informações na rede.

A Tabela 7.1 apresenta os trabalhos relacionados nesta seção, descrevendo os mecanismos de sincronização utilizados por cada solução. Vale ressaltar que a principal diferença no presente trabalho pode ser visualizada na coluna que especifica o local de sincronização. Note que a proposta neste trabalho executa a sincronização no plano de controle sem a participação direta das entidades da rede SDN, como exemplo os *switches* e os controladores. Mais detalhes da abordagem utilizada são apresentados na próxima seção.

[1]https://www.opendaylight.org/

[2]http://infinispan.org/

Tabela 7.1: Trabalhos relacionados à sincronização de dados em redes SDN.

Trabalhos Relacionados	Algoritmo de Sincronização	Plano de Sincronização	Local de Sincronização dos Algoritmos
[Schiff et al., 2016]	Transações atômicas usando *compare-and-set* (CAS)	Plano de Dados	Switches SDN
[Dang et al., 2015]	Algoritmo de consenso *NetPaxos*	Plano de Dados	Switches SDN
[Ho et al., 2016]	Algoritmo de consenso *Fast Paxos-based Consensus*	Plano de Controle	Controladores SDN
[Koponen et al., 2010]	Solução de consenso *Zookeeper*	Plano de Controle	Controladores SDN
[Canini et al., 2015]	Algoritmo *Policy Serialization* (PS) baseado em Rep. de Máquina de Estados	Plano de Controle	Controladores SDN
Plataforma ODL	Algoritmo de consenso Raft	Plano de Controle	Controladores SDN

7.3 Um Plano de Controle Robusto Baseado em VNF

Nesta seção é apresentada uma caracterização do problema abordado: a consistência do plano de controle em redes SDN. Na sequência é detalhada a arquitetura e as particularidades de implementação, bem como, uma breve descrição do algoritmo de consenso Paxos e detalhes de como sua lógica é implementada e executada na *VNF-Consensus*.

7.3.1 Caracterização e Delimitação do Problema

Para garantir um plano de controle consistente em uma rede SDN, todos os eventos a serem atualizados precisam ser sincronizados entre os controladores envolvidos. Cada atualização gera um novo estado da rede. As mudanças de estados são causadas por eventos como, por exemplo: a criação de um novo fluxo de comunicação, a ocorrência de falhas em *links* ou *switches*, a instalação de regras para passar por serviços como filtros de pacotes ou novas rotas, entre outros. Em síntese, neste contexto é necessário utilizar algum mecanismo que sincronize cada evento evitando estados inconsistentes entre os diversos controladores. Os autores de [Schiff et al., 2016] descrevem diversas patologias causadas na rede devido a esses estados inconsistentes.

As decisões realizadas no plano de controle implicam em mudanças no plano de dados, isto é, a criação de um novo fluxo de dados implicará na instalação de uma nova regra no *switch*. Portanto, é importante entender a comunicação entre os controladores e os *switches* SDN. Em [Tianzhu et al., 2016] os autores identificam dois tipos de comunicação (ver Figura 7.1): *Switch-to-Controller* e *Controller-to-Controller*, descritos a seguir.

A comunicação *Switch-to-Controller* suporta iterações entre os *switches* e o controlador usando o protocolo *OpenFlow*. Por exemplo, é nesse plano que um pacote *packet-in* (pacote gerado quando não há uma correspondência na tabela de fluxos do *switch*) é encaminhado do *switch* ao controlador. O controlador, por sua vez, retorna uma mensagem *flow-mod* para autorizar ou não a comunicação do pacote. Essas são mensagens utilizadas para gerenciar as tabelas de fluxos de cada *switch*. Considerando a execução de múltiplos controladores, a decisão tomada para a autorização do novo fluxo de pacotes deve ser sincronizada entre todos os controladores da rede. Caso contrário, estados inconsistentes no plano de controle poderão causar patologias na rede, como o descarte de pacotes, criação de *loops*, rotas contornando serviços específicos, entre outros. Além disso, a mensagem *flow-mod* precisa ser atualizada pelos *switches* no mesmo instante de tempo (no contexto do plano de dados), evitando estados

inconsistentes. Entretanto, esta tarefa pode ser significativamente complicada em ambientes onde os atrasos não podem ser previstos [Zhou et al., 2014]. Portanto, vale ressaltar que neste trabalho abordamos a consistência no plano de controle, ao passo que a sincronização no plano de dados pode ocorrer em instantes diferentes de tempo. Em resumo, assumimos que nos estados transientes do plano de dados há a possibilidade de ocorrer períodos com estados inconsistentes da rede.

A comunicação *Controller-to-Controller* permite a sincronização do plano de controle. A consistência nesse plano exige que todos os controladores possuam a mesma visão do estado da rede. No plano de controle a consistência pode ser forte ou eventual [Tianzhu et al., 2016]. A consistência forte significa que as leituras dos dados em diferentes controladores produzem sempre o mesmo resultado, ao passo que, a consistência eventual significa que há períodos transientes em que as leituras em diferentes controladores podem produzir diferentes valores. No presente trabalho utilizamos o algoritmo de consenso Paxos que implementa a consistência forte.

7.3.2 VNF-Consensus

Em uma rede SDN as ações executadas por um conjunto de controladores precisam ser sincronizadas e coordenadas. Como mencionado acima, a *VNF-Consensus* implementa um algoritmo de consenso que possibilita garantir a consistência do plano de controle em redes SDN. Em particular, a *VNF-Consensus* implementa, no ambiente de rede, o algoritmo de consenso Paxos descrito a seguir. Note que, embora sucinta, a descrição do algoritmo é importante para a compreensão da *VNF-Consensus* que será detalhada na sequência.

Paxos é um algoritmo de consenso tolerante a falhas proposto por Leslie Lamport [Lamport, 1998]. Na Seção 3.2 desta tese o Paxos foi brevemente descrito. Recapitulando, informalmente, um algoritmo de consenso [Santos et al., 2011] permite que um conjunto de processos entre em acordo sobre um determinado valor, sendo que inicialmente cada processo pode propor valores distintos. Dentre as propriedades que um algoritmo de consenso deve atender, destaca-se neste trabalho duas propriedades: segurança (*safety*) e progresso (*liveness*). A segurança garante que todos os processos sem falha decida pelo mesmo valor. O progresso garante que o consenso termine com sucesso. No algoritmo Paxos os processos podem assumir três diferentes papéis denominados de: *proposers*, *acceptors* e *learners*.

Os *proposers* propõem um valor, os *acceptors* escolhem um valor e os *learners* aprendem o valor decidido. Um único processo pode assumir qualquer uma dessas funções e múltiplas funções simultaneamente. Paxos é ótimo em termos de resiliência [Lamport, 2006b]: para tolerar f falhas ele requer $2f + 1$ *acceptors* – isto é, para assegurar o progresso um quórum de $f + 1$ *acceptors* devem estar sem-falha. Para resolver o consenso, Paxos executa em rodadas. Tipicamente um *proposer* ou um *acceptor* atua como *coordenador* da rodada. O *coordenador* recebe propostas que tentam alcançar o consenso. Quando o quórum de *acceptors* aceita o mesmo número de rodada, o consenso termina. Neste momento os *learners* têm acesso ao valor decidido.

Paxos é utilizado neste trabalho para sincronizar os dados no plano de controle. Em especial, os controladores aprendem (*learners*) os valores decididos e fornecidos pela *VNF-Consensus*. Dessa maneira, o plano de controle mantém o estado da rede em todos os controladores. Paxos é baseado em um modelo de consistência forte. Portanto, a *VNF-Consensus* garante que os dados resultantes das leituras de diferentes controladores irão produzir sempre o mesmo resultado.

Para obter uma visão comum do estado da rede, cada controlador possui acesso a uma instância da *VNF-Consensus* através da qual poderá receber decisões e enviar dados para serem sincronizados. A Figura 7.1 apresenta a integração dos controladores SDN com a *VNF-*

Consensus. Todas as decisões realizadas pela *VNF-Consensus* são sistematicamente executadas sem a atuação direta dos controladores. Assim, cada controlador é um cliente do algoritmo de consenso que irá aprender os resultados gerados pela *VNF-Consensus*.

Considere que o *host x* apresentado na Figura 7.1 deseja, pela primeira vez, fazer uma comunicação com outro dispositivo qualquer. O *host x* encaminha o pacote ao *switch OpenFlow* que, por sua vez, verifica se há regras deste destinatário em sua tabela de fluxos. Se nenhuma entrada existir na tabela de fluxos referente ao destinatário, o *switch* encaminha um *packet-in* ao controlador SDN. Antes do controlador decidir o que fazer com o pacote, ele encaminha uma cópia do mesmo para a *VNF-Consensus*. O consenso então é inicializado e, ao final, o resultado é encaminhado a todos os controladores SDN.

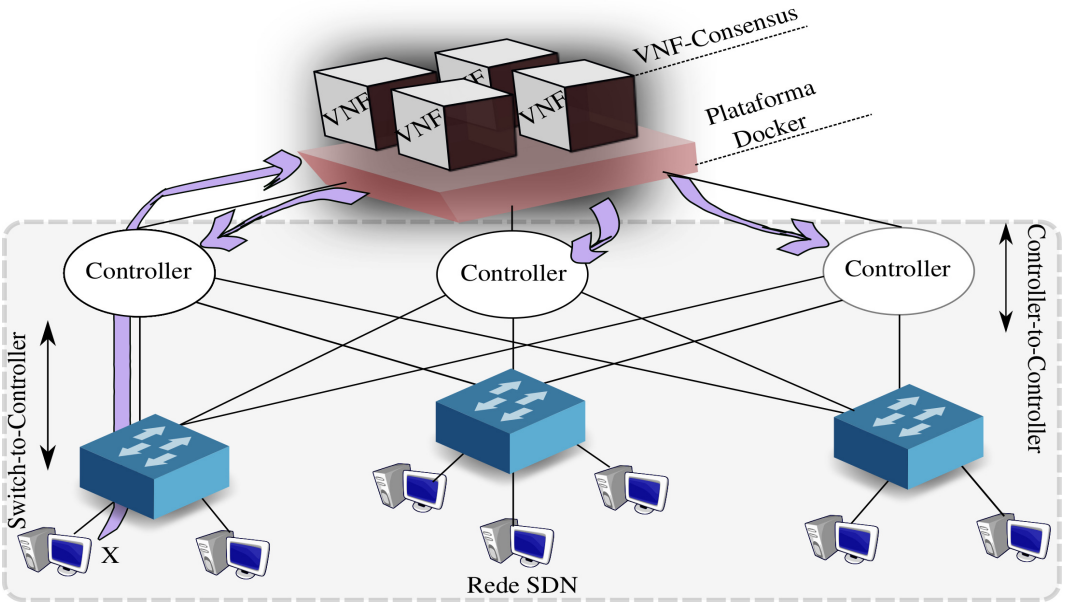


Figura 7.1: Integração da VNF na arquitetura do controlador SDN.

Observe que quando o controlador SDN recebe um novo fluxo, como no exemplo apresentado no parágrafo anterior, faz-se necessário executar um 'desvio' no fluxo dos pacotes com o objetivo de encaminhá-los à *VNF-Consensus*. Em outras palavras, cada pacote precisa ser classificado. A classificação permite definir qual pacote será encaminhado e para qual destino. Se for decidido que um pacote precisa ser encaminhado para a *VNF-Consensus*, então um fluxo de encaminhamento precisa ser criado.

Para executar essa tarefa é proposta uma ferramenta denominada de *Filter*. A ferramenta *Filter* realiza a classificação e, quando necessário, o encaminhamento dos pacotes para a *VNF-Consensus*. O funcionamento e a integração desses componentes podem ser visualizados na arquitetura apresentada na Figura 7.2. Observe que as instâncias da *VNF-Consensus* são executadas no ambiente de virtualização como *containers*.

Note (na arquitetura apresentada na Figura 7.2), que a ferramenta *Filter* é composta por dois módulos denominados de *Classifier* e *Forwarder*. O módulo *Classifier* possui acesso às regras que são armazenadas no repositório *Rules*. Essas regras permitem classificar os fluxos de entrada para determinar o destino de cada pacote. O processo ocorre de acordo com os passos mostrados na Figura 7.2 e detalhados a seguir: o controlador SDN encaminha todos os pacotes para a ferramenta *Filter* (passo 1), se houver alguma regra cadastrada no módulo *Classifier* que se encaixe ao pacote recebido (passo 2), o pacote é repassado para o módulo *Forwarder* (passo 4), caso contrário, o pacote será retornado ao controlador (passo 3). O *Forwarder*, por sua vez, é

responsável por realizar o repasse dos pacotes, por exemplo, permitindo encaminhar o pacote à correspondente VNF (passo 5). Esse último passo também é identificado como SFP (*Service Function Path*).

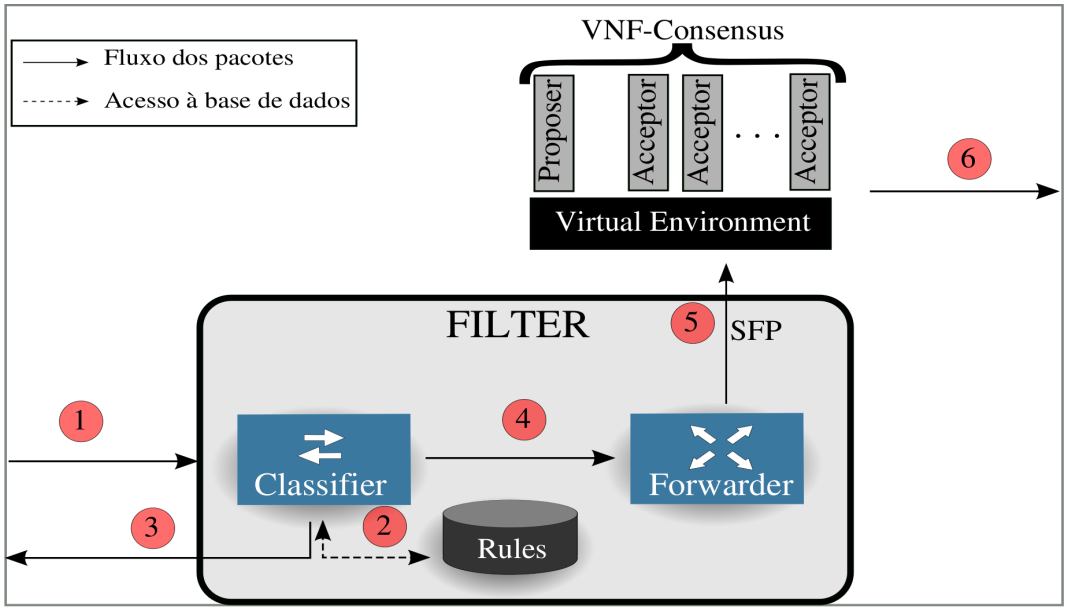


Figura 7.2: Arquitetura para a sincronização do plano de controle.

Quando um SFP é criado, o pacote contendo a regra a ser instalada é repassado para o coordenador da *VNF-Consensus*. Do ponto de vista do Paxos, neste cenário cada VNF executa funções distintas de acordo com as especificações do algoritmo. Isto é, na arquitetura apresentada na Figura 7.2 considere o *proposer* como sendo também o coordenador. É o coordenador quem recebe as regras vindas dos diversos controladores da rede. Ao receber uma regra, o coordenador então inicia uma nova instância de consenso. Uma instância é definida executando duas fases distintas do algoritmo Paxos. Na primeira fase, o coordenador seleciona um número único para a rodada e o envia em uma solicitação para os *acceptors*. Ao receber a solicitação com um número maior que qualquer outro recebido previamente para aquela instância, o *acceptor* responde ao coordenador prometendo que rejeitará qualquer solicitação futura com um número de rodada menor. Se o *acceptor* já aceitou uma regra para a instância, ele retorna essa regra ao coordenador junto com o número de rodada recebido quando a regra foi aceita. Quando o coordenador recebe respostas de um quórum de *acceptors*, a segunda fase do protocolo é inicializada.

Na segunda fase, o coordenador verifica os valores recebidos do quórum de *acceptors*. Se não houver *acceptors* que tenham aceitado uma regra, o coordenador executa a fase 2 com sua regra. Por outro lado, se algum *acceptor* retornou uma regra na primeira fase, o coordenador é obrigado a executar a segunda fase com a regra que possuir o maior número de rodada. Com a regra selecionada, o coordenador envia uma requisição aos *acceptors* com o número da rodada e a regra escolhida. Se os *acceptors* não prometeram participar de uma rodada com número maior, então eles respondem ao coordenador aceitando a regra proposta. Quando o coordenador recebe respostas de um quórum de *acceptors* há a garantia de que não há outro *proposer*, ou coordenador, com um número de proposta maior que a já aceita. A regra então é enviada aos *learners* e o consenso para aquela instância termina.

O coordenador envia o valor decidido aos *learners* que são controladores SDN que, dessa forma, aprendem o valor decidido pela *VNF-Consensus*.

Note que para cada papel definido no algoritmo Paxos um *container* é criado para ser executado de forma independente entre os processos. Essa abordagem permite o isolamento total dos processos. A implementação em *containers* é motivada por sua baixa utilização dos recursos computacionais, rápida instanciação e fácil implementação de funções virtualizadas, possibilitando também uma alta densidade de processos virtualizados no mesmo *host* [Anderson et al., 2016]. A seguir são apresentados os experimentos que permitem comprovar os benefícios do serviço descrito nesta seção.

7.4 Avaliação da NFV-Consensus

Nesta seção são apresentados experimentos executados para avaliar o desempenho da *VNF-Consensus* em uma rede SDN. O ambiente para execução dos experimentos é implementado usando o protocolo *OpenFlow* e controladores *Ryu*[3]. O ambiente é hospedado em uma máquina física com as seguintes características: processador AMD FX-4300 CPU 3.8GHz com 4 núcleos e sistema operacional *Ubuntu* 16.04 com *kernel* 4.4.0-53.

Os *switches* SDN foram simulados através da ferramenta *cbench*[4]. *cbench* é uma ferramenta para avaliação de desempenho de controladores SDN que permite gerar altas taxas de fluxos OpenFlow na rede. *cbench* emprega OVS (Open VSwitch[5]) para a simulação dos *switches*. Cada instância da *VNF-Consensus* é hospedada em um *container* particular, bem como o próprio controlador Ryu. Para a sincronização dos dados é utilizada uma biblioteca Paxos denominada de libPaxos[6]. Vale ressaltar que a solução proposta pode fazer uso de qualquer algoritmo de consenso.

A seguir são apresentados três grupos de experimentos. No primeiro grupo o objetivo é avaliar e comparar o impacto causado para a sincronização do plano de controle. No segundo grupo é avaliado o *throughput* do consenso comparando sua execução como uma VNF e através de sua execução nos próprios controladores.

Por fim, no terceiro grupo é avaliado o comportamento da solução proposta de acordo com o aumento do número de controladores SDN. Os experimentos possuem configurações específicas que são descritas respectivamente dentro de cada grupo.

Custo para a sincronização do plano de controle

Quando a sincronização do plano de controle é deixada a cargo dos próprios controladores SDN, aumenta-se a quantidade de tarefas que os controladores precisam executar. Nestas condições, pode-se ter um impacto negativo no desempenho global da rede. Para medir tal impacto, a Figura 7.3 apresenta os seguintes experimentos: a utilização de CPU (Figura 7.3(a)), o número de fluxos por segundo tratado pelo controlador (Figura 7.3(b)) e o tempo para um controlador instalar um conjunto de regras (Figura 7.3(c)).

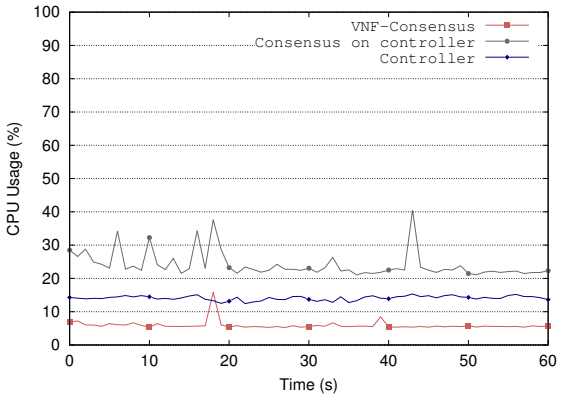
O objetivo do experimento apresentado na Figura 7.3(a) é medir a carga extra de trabalho, considerando a utilização de CPU, causada pelas ações de sincronização realizadas pelos próprios controladores. A utilização de CPU apresentada na figura 7.3(a) é descrita em 3 cenários: 1) o controlador executa suas operações regulares enquanto a *VNF-Consensus* é responsável por manter o plano de controle consistente (curva Controlador). Nesta situação o controlador somente repassa as regras para a *VNF-Consensus*. Vale ressaltar que o controlador

[3]https://osrg.github.io/ryu/

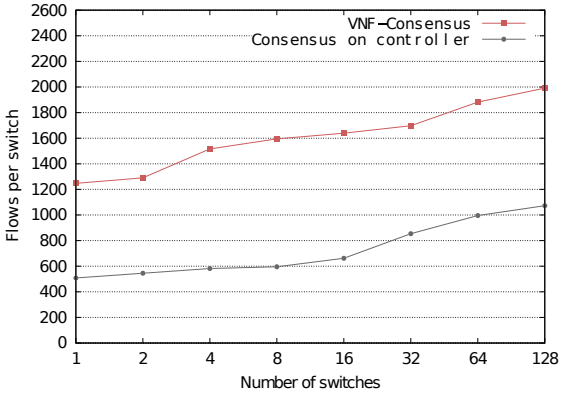
[4]https://github.com/mininet/oflops/tree/master/cbench

[5]https://openvswitch.org

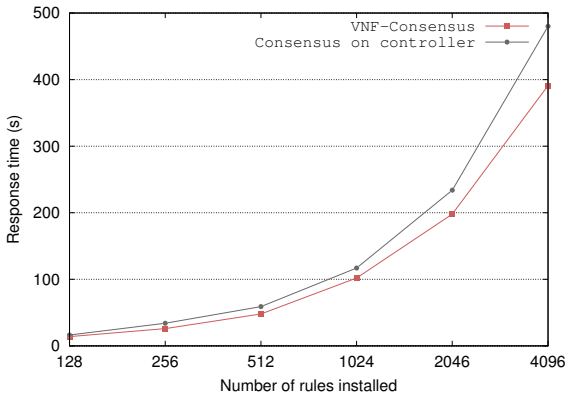
[6]https://bitbucket.org/sciascid/libpaxos



(a) Uso de CPU.



(b) Fluxo por segundo tratado pelo controlador.



(c) Tempo necessário para instalar um conjunto de regras.

Figura 7.3: Sincronização do plano de controle: avaliação de desempenho.

não fica bloqueado aguardando a resposta da *VNF-Consensus*; 2) Paxos está sendo executado em um controlador SDN (curva Consenso no controlador): o controlador é responsável por todas as atividades, incluindo a execução do consenso; 3) Paxos está sendo executado na *VNF-Consensus* (curva *VNF-Consensus*). Para cada experimento três amostras foram coletadas, cada uma de 60 segundos. A topologia de rede é composta por três controladores e três instâncias da *VNF-Consensus*.

Para este experimento, os controladores foram submetidos a um fluxo de dados gerado a partir dos *hosts* através da ferramenta *cbench*. Note que quando o próprio controlador é utilizado para executar o consenso, o uso de CPU é de aproximadamente 27,1%. Por outro lado, quando é utilizada a *VNF-Consensus* para sincronizar o plano de controle, a carga de trabalho dos controladores cai drasticamente para 14,3%. A utilização da CPU pela *VNF-Consensus* ficou aproximadamente 16%. Este experimento claramente mostra as vantagens de desacoplar o algoritmo de consenso dos controladores. Como resultado, percebe-se que o controlador permanece mais disponível para executar suas tarefas regulares no ambiente de rede. A razão é que o custo de sincronização é realocado para uma entidade externa, ou seja, a própria *VNF-Consensus*.

A Figura 7.3(b) mostra outra vantagem da *VNF-Consensus*. O propósito deste experimento é avaliar o impacto no controlador quando ele precisa gerenciar uma rede SDN (por exemplo, instalar uma regra no *switch* SDN) e, em paralelo, necessita sincronizar o plano de controle. No experimento utiliza-se três controladores, três VNFs e o número de *switches* é

aumentado gradativamente, iniciando com 1 até 128. Cada experimento possui a duração de 1 minuto. Conforme aumentamos o número de *switches* o experimento é repetido. Como resultado, um grande número de fluxos é criado. Por exemplo, cada um dos 8 *switches* mostrados na Figura 7.3(b) encaminha 600 fluxos por segundo (fluxos/s), o total de fluxos processados pelo controlador é de 4800 fluxos/s. Assim, no eixo y é mostrado o fluxo por segundo por *switch*. Note na Figura 7.3(b) que quando a *VNF-Consensus* executa a sincronização, o controlador é capaz de processar um grande número de pacotes. A razão para que isso aconteça é que o controlador tem uma carga de trabalho menor do que quando ele precisa executar em paralelo a sincronização do plano de controle. A diferença é próxima de 53% e assim permanece ao longo do eixo x.

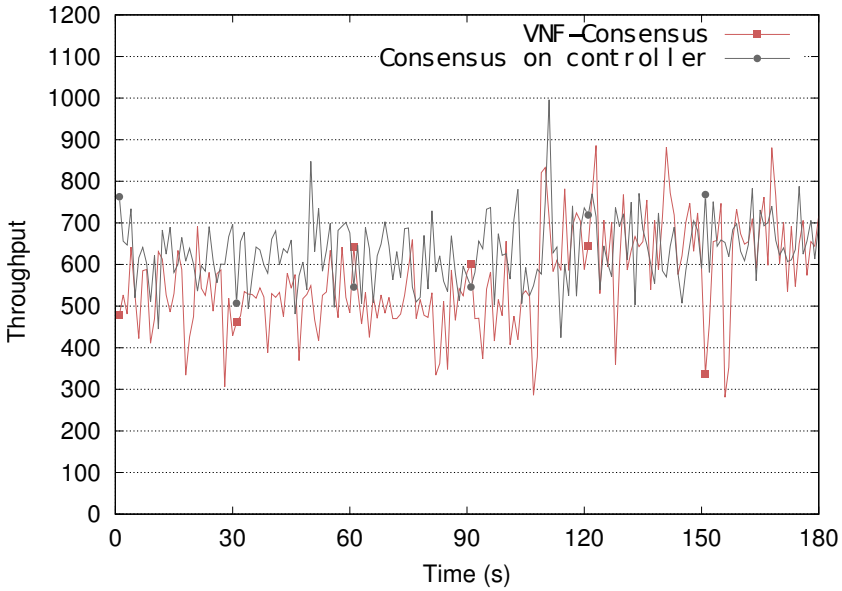
Na terceira parte deste experimento apresentado na Figura 7.3(c), um *script* foi criado para produzir, de maneira contínua, requisições REST que instalam regras aleatórias no *switch*. Quando um controlador recebe uma requisição ele instala um grupo de regras no *switch*. Neste momento é medido o tempo para executar a tarefa para a instalação da regra, ou seja, é medido o tempo desde o início da requisição até o seu respectivo retorno, que ocorre quando o *switch* instala a regra e retorna uma confirmação ao controlador (Host –> Switch –> Controller –> Switch –> Host). Em paralelo, a ferramenta *cbench* envia um fluxo de dados fazendo com que o plano de controle seja constantemente sincronizado. Neste experimento utilizou-se três controladores e três VNFs. Vale ressaltar que esse tempo é medido desde o início da requisição até o seu respectivo retorno, isto é, quando o *switch* instala a regra e retorna uma confirmação ao controlador. Cada controlador gerencia um único *switch*. Na Figura 7.3(c) conforme aumenta-se o número de regras a serem instaladas o tempo de resposta também aumenta. Entretanto, observa-se que a curva correspondente a *VNF-Consensus* é menor, reduzindo o tempo de resposta em até 18,5%. Isto significa que a estratégia proposta supera as abordagens que adotam a sincronização nos controladores.

Análise do throughput do Paxos

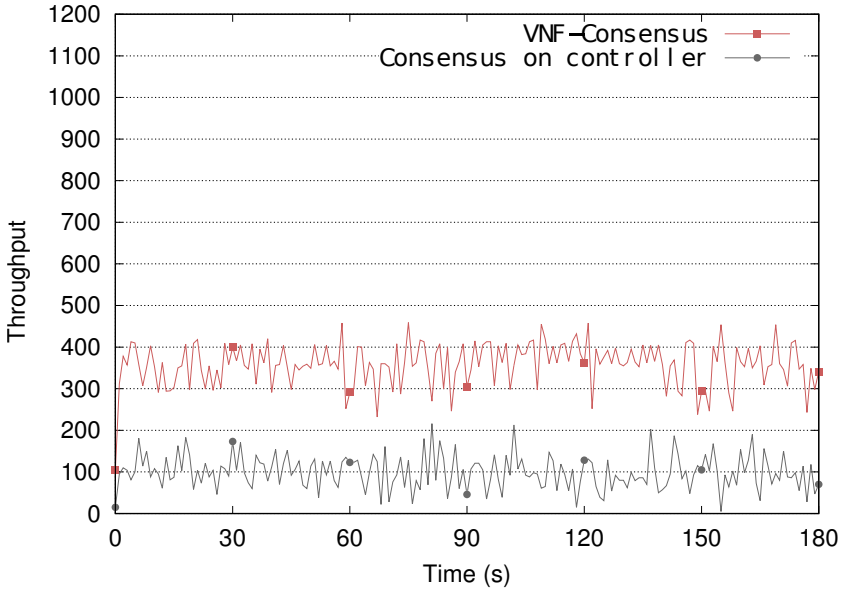
Nesta série de experimentos é medido o *throughput* do consenso quando o Paxos é executado como uma instância da VNF, ou quando ele está sendo executado no controlador. A topologia desses experimentos consiste de três controladores, três VNFs e três *switches* (um *switch* por controlador). O eixo y do gráfico da Figura 7.4 mostra o número de execuções de consenso processadas por segundo durante um período de 180 segundos (eixo x).

Na Figura 7.4(a) são submetidas continuamente requisições para a sincronização do controlador. Neste cenário, o controlador não executa nenhuma outra tarefa em paralelo, isto é, o controlador é dedicado a executar o consenso depois de cada requisição de atualização. Neste experimento é possível notar que a VNF possui um menor *throughput*. Este resultado é de certo modo esperado, pois as instâncias das VNFs estão executando em *containers* localizados fora dos controladores. Ao inicializar a sincronização, os controladores executam requisições para serem sincronizadas na *VNF-Consensus*, que por sua vez retorna a decisão para os controladores. Assim, a execução do consenso pela *VNF-Consensus* apresenta um menor *throughput* devido a este passo extra na comunicação entre o controlador e a VNF. Paxos baseado no controlador tem um *throughput* de 11,4% melhor do que a *VNF-Consensus*.

Em contraste, no experimento mostrado na Figura 7.4(b), o controlador executa tarefas em paralelo, ou seja, executa tarefas de sincronização, como também forçam o controlador atualizar a tabela de fluxos do *switch*. Como resultado, é possível notar que o *throughput* cai significativamente em ambos os casos comparados. Entretanto, o *throughput* da *VNF-Consensus* é 3,6 vezes maior do que o Paxos executando no controlador. Isso significa que novamente a



(a) Medida do *throughput* sem incluir tarefas em paralelo nos controladores.



(b) Medida do *throughput* incluindo tarefas em paralelo nos controladores.

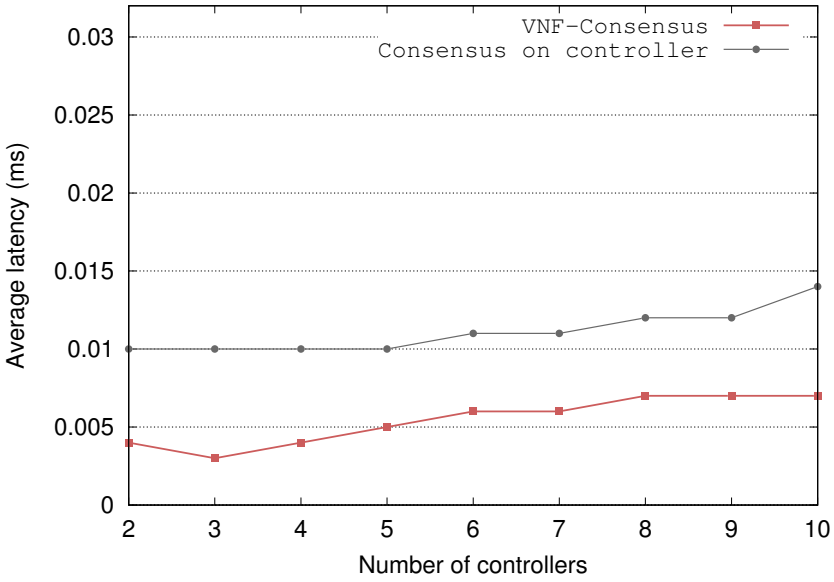
Figura 7.4: Comparação do *throughput*.

VNF-Consensus mostrou ser uma solução eficiente, mesmo em cenários com sobrecarga de trabalho.

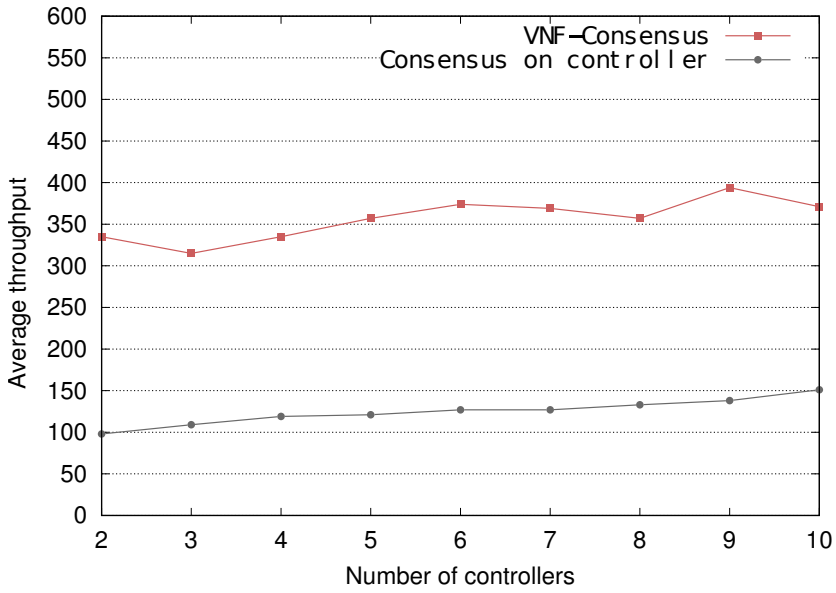
Aumentando o Número de Controladores

Neste último experimento é investigada a latência e, novamente, o *throughput* do consenso. Neste caso, aumenta-se gradativamente o número de controladores. A topologia de rede consiste em um *switch* por controlador e três instâncias da *VNF-Consensus*. Ao passo que, o número de instâncias Paxos no controlador é proporcional ao número de controladores.

A Figura 7.5(a) mostra a latência do Paxos quando o algoritmo é executado nos controladores e nas instâncias da *VNF-Consensus*. Este experimento mostra que a *VNF-Consensus*



(a) Média da latência do Paxos.



(b) Média do *throughput* do Paxos por segundo.

Figura 7.5: Aferição da escalabilidade.

apresenta baixa latência. Enquanto que o Paxos executado nos controladores apresenta uma média de latência de aproximadamente 0,011ms, ao passo que a latência apresentada pela VNF-Consensus é de aproximadamente 0,005ms. Assim, a *VNF-Consensus* reduz a latência do Paxos em aproximadamente 54%. Além disso, note que o número de instâncias da *VNF-Consensus* é configurável e independente do número de controladores. Em outras palavras, a latência do Paxos quando executado no controlador aumenta quase linearmente com o número de controladores.

Na Figura 7.5(b) comparamos a *VNF-Consensus* com o Paxos nos controladores em termos do número de execuções realizadas por segundo, enquanto o número dos controladores cresce até 10. Como pode ser observado, a *VNF-Consensus* possui um *throughput* aproximadamente 3 vezes maior. Relembre que o número de instâncias da *VNF-Consensus* permanece em 3.

Portanto, é importante notar que independente do número de controladores, a *VNF-Consensus* sempre apresenta um melhor *throughput*.

Por fim, ao analisarmos todos os resultados realizados neste trabalho podemos concluir que quando a sincronização do plano de controle é realizada pela *VNF-Consensus*, tem-se uma redução na carga dos controladores, o que tem impacto significativo na escalabilidade, na medida em que há um claro limite para a quantidade de tarefas que um controlador pode executar antes de se tornar um gargalo da rede.

7.5 Considerações Parciais

Neste capítulo foi proposta uma solução para a sincronização do plano de controle em redes SDN, na qual um grupo de controladores distribuídos se mantêm consistentes com o auxílio de uma função virtualizada de rede denominada de *VNF-Consensus*. A *VNF-Consensus* implementa o algoritmo de consenso Paxos no ambiente de rede. Dessa forma, todas as decisões realizadas pela *VNF-Consensus* são executadas sem a atuação direta dos controladores. Os resultados experimentais mostraram que quando os próprios controladores executam a sincronização do plano de controle, há um impacto negativo no desempenho global da rede. Por outro lado, com a utilização da *VNF-Consensus*, o plano de controle é sincronizado sem aumentar a carga de trabalho nos controladores. Por consequência, vimos que há melhorias no desempenho da rede. Vale ressaltar que a estratégia proposta se difere das demais encontradas na literatura por não estar implementada dentro dos *switches* e, nem sequer, nos controladores SDN.

Capítulo 8

AnyBone

Este capítulo descreve a proposta de uma função virtualizada de rede denominada de *AnyBone* a qual oferece primitivas de difusão confiável para garantir a entrega, inclusive ordenada, das mensagens transmitidas na rede. Para realizar tal função, o *AnyBone* divide a responsabilidade entre dois componentes que operam integrados. Um dos componentes é uma Função Virtualizada de Rede denominada de *VNF-Sequencer* localizada dentro da rede SDN. Em especial, a *VNF-Sequencer* garante a ordem total das mensagens entregues entre todos os processos por utilizar um sequenciador que gerencia as transmissões. O outro componente é uma biblioteca denominada de *RBCast* e oferece uma API para as aplicações trocarem mensagens utilizando as primitivas de difusão confiável.

A justificativa detalhada da presente proposta é apresentada na próxima seção. Na Seção 8.2 são apresentados os detalhes de funcionamento dos algoritmos de difusão confiável e para a entrega ordenada das mensagens, a arquitetura e a lógica de implementação executada pelo *AnyBone* também são descritas. Por fim, na Seção 8.3 são realizados os experimentos que comprovam e permitem demonstrar as funcionalidades implementadas pelo *AnyBone*, as considerações parciais são apresentadas na Seção 8.4.

8.1 Justificativa da Proposta

Na Seção 3.3 foram apresentados os conceitos sobre difusão confiável. Relembrando, a difusão confiável pode ser considerada como um caso especial de comunicação que ocorre quando um processo deseja transmitir uma mensagem que precisa ser entregue, garantidamente, a todos os demais processos corretos do sistema [Défago et al., 2004]. Essa comunicação, conhecida também como *reliable broadcast*, é fundamental para a construção de aplicações distribuídas tolerantes a falhas [Pedone e Schiper, 2003]. Além disso, vimos também que a comunicação pode exigir que todas as mensagens sejam entregues de forma ordenada, essa primitiva denomina-se difusão atômica confiável (*atomic reliable broadcast*).

Na prática, a implementação das primitivas citadas exige um grande esforço para a coordenação entre os processos, aumentando a complexidade para a implementação de aplicações distribuídas [Li et al., 2016], haja vista que, em geral, esse tipo de serviço precisa ser garantido pela própria aplicação ou pelo uso de serviços adjacentes [Défago et al., 2004][1]. Uma outra abordagem que se mostra bastante atraente é fazer com que a própria rede de comunicação forneça primitivas que permitem garantir a entrega ordenada das mensagens para as aplicações.

[1]Défago apresenta uma lista de serviços oferecidos como um *Middleware* que possibilita garantir a difusão confiável

No trabalho apresentado em [Li et al., 2016], os autores utilizam uma nova abordagem para garantir a replicação consistente em *data centers* através de uma solução que divide as obrigações entre a rede e a aplicação. A rede garante a ordem dos pacotes, enquanto que o protocolo de replicação garante a entrega das mensagens. O protocolo utilizado é denominado de NOPaxos (*Network-Ordered Paxos*). NOPaxos é executado somente quando necessário, evitando a sincronização constante entre os processos. Em outras palavras, quando ocorrem imprevistos na comunicação, por exemplo, uma mensagem é perdida, NOPaxos deve ser executado explicitamente. A ordem das mensagens é garantida pela implementação de um sequenciador localizado dentro da rede. Os autores mostram que quando o sequenciador é implementado diretamente nos *switches*, a replicação ocorre com baixa latência e baixo *throughput*, fornecendo uma replicação praticamente sem custo.

Neste contexto, o presente capítulo descreve uma função virtualizada de rede denominada de *AnyBone* a qual oferece as primitivas de difusão confiável dentro de uma rede SDN. Isto é, a própria rede oferece uma função que possibilita a entrega confiável das mensagens e de maneira ordenada. Em especial, *AnyBone* garante a ordem das mensagens por um sequenciador localizado dentro da rede, enquanto que a entrega é realizada por uma biblioteca localizada nos processos finais. Sendo assim, *AnyBone* é composta por uma Função Virtualizada de Rede denominada de *VNF-Sequencer* e uma biblioteca denominada de *RBCast*. *RBCast* oferece uma API para as aplicações trocarem mensagens através das seguintes primitivas: *reliable broadcast*, *atomic reliable broadcast*, *atomic FIFO reliable broadcast* e *atomic causal reliable broadcast*. Como resultado, *AnyBone* é responsável por realizar e gerenciar as trocas de mensagens de maneira a garantir as propriedades da comunicação definida pela aplicação distribuída.

8.2 AnyBone: Uma Rede com Difusão Confiável e Ordenada de Mensagens

Nesta seção são apresentados os algoritmos de difusão confiável. A arquitetura e a lógica de implementação modular utilizada no *AnyBone*, também são descritos.

8.2.1 Difusão Confiável de Mensagens pelo AnyBone

AnyBone é uma NFV que provê primitivas de comunicação para a transmissão de mensagens via difusão confiável e ordenada. *AnyBone* oferece vários algoritmos clássicos para garantir ambos, a entrega e ordem total das mensagens. Estes algoritmos são executados de forma modular, organizado de forma similar a uma pilha. Por exemplo, para a entrega confiável das mensagens, é necessário sempre executar o algoritmo para a difusão confiável, ao passo que, para garantir também a ordem total das mensagens, é necessário executar também, no topo da difusão confiável, o algoritmo de difusão atômica. Neste formato modular os algoritmos são simplificados e mais fáceis de entender [Hadzilacos e Toueg, 1994].

Para garantir a entrega das mensagens é necessário realizar a comunicação usando algum algoritmo para a difusão confiável. Uma versão simplificada deste algoritmo é proposta por Chandra e Toueg [Chandra e Toueg, 1996] e seu pseudo código é mostrado no Algoritmo 4. Este algoritmo garante a entrega das mensagens da seguinte forma: quando um processo recebe uma mensagem *m* por difusão confiável, pela primeira vez, ele retransmite *m* para todos os processos e então entrega a mensagem para a aplicação. Cada mensagem, na difusão confiável, é composta por uma estrutura que descreve o emissor da mensagem e o conteúdo do pacote.

Algorithm 4: Algoritmo para difusão confiável.

```
1 Todo processo p_i executa como segue:

2 Para executar R_broadcast(m):
3     envia m para todos (incluindo p_i)

4 R_deliver(m) ocorre da seguinte forma:
5     Quando recebe m pela primeira vez
6     se (emissor (m) ≠ p_i) então
7         envia m para todos
8         R_deliver(m)
```

O *AnyBone* garante a difusão confiável implementando o Algoritmo 2 que foi apresentado na Seção 5.3.1. O Algoritmo 2 difere-se do algoritmo apresentado nesta seção por fazer uso de um detector de falhas. A difusão confiável implementada no *AnyBone* é avaliada nos experimentos que são apresentados na Seção 8.3. Na próxima seção é apresentado detalhes de como o *AnyBone* constrói a ordem total das mensagens transmitidas por difusão para serem entregues às aplicações.

8.2.2 Construindo a Ordem das Mensagens Entregues pelo AnyBone

Em um sistema que precisa manter a ordem total dos eventos, o problema que precisa ser resolvido é: "Como construir a ordem?". Na literatura existem algumas alternativas para a implementação da ordem total dos eventos como, por exemplo, a utilização de um sequenciador que pode ser fixo, móvel, baseado em privilégios, entre outros [Défago et al., 2004]. No *AnyBone* a ordem das mensagens é implementada através de um processo específico que recebe o papel de sequenciador (também denominado de *sequencer*). O sequenciador é responsável por construir a ordem total das mensagens.

Neste esquema, o processo que recebe o papel de sequenciador é único e, em geral, a responsabilidade não é transferida para outro processo. A ordem total é construída da seguinte forma: para uma mensagem m ser transmitida por difusão, primeiramente o emissor de m encaminha a mensagem para o sequenciador, quando m é recebida pelo sequenciador a mensagem obtém um valor único de sequência atribuído pelo próprio sequenciador, então m é transmitida por difusão para todos os destinatários. Por fim, os processos receptores entregam m de acordo com o valor de sequência atribuído pelo sequenciador.

O sequenciador fixo pode ser implementado através de 3 métodos apresentados na Figura 8.1 [Défago et al., 2004]: UB (*Unicast-Broadcast*), BB (*Broadcast-Broadcast*) e UUB (*Unicast-Unicast-Broadcast*).

A Figura 8.1(a) apresenta o método UB onde o processo que deseja transmitir uma mensagem por difusão (*sender*) executa uma comunicação *unicast* (ponto-a-ponto) seguida de um *broadcast* executado pelo *sequencer*. O *sequencer*, ao executar o *broadcast*, adiciona um número de sequência na mensagem. Do lado do receptor, a mensagem será entregue para a aplicação respeitando a ordem indicada pelo *sequencer*. No método BB apresentado na Figura 8.1(b), o *sender* executa um *broadcast* para todos os destinatários incluindo o *sequencer*. O *sequencer* adiciona um número de sequência na mensagem e executa um *broadcast* para todos os destinatários. Note que esse método gera um número maior de mensagens do que o método UB,

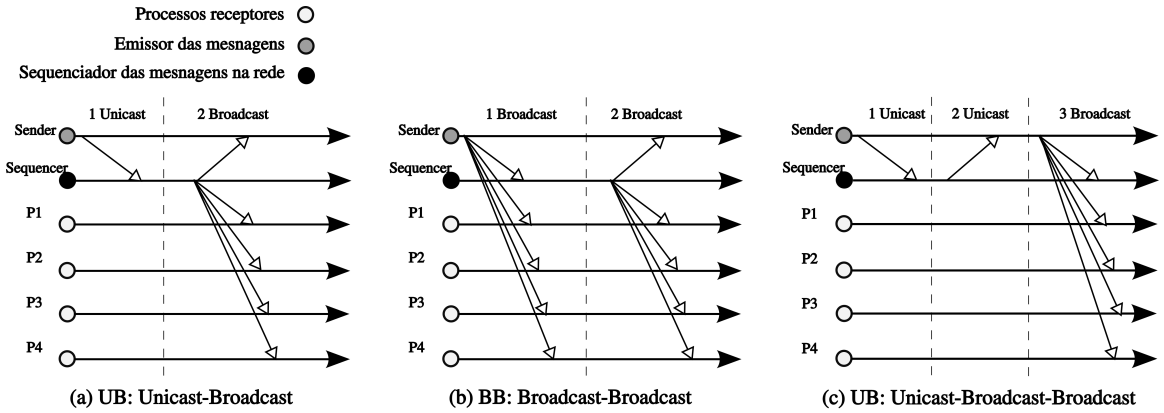


Figura 8.1: Três métodos para implementar o sequenciador fixo.

por outro lado, o método BB facilita a implementação de um *sequencer* tolerante a falhas. Por fim, no método UUB apresentado na Figura 8.1(c), o *sender* solicita um número de sequência para o *sequencer*, então o próprio *sender* inclui o número de sequência na mensagem e a transmite para todos os processos. Este último método é menos comum que os demais e inclui um terceiro passo que o torna menos atraente [Défago et al., 2004].

No *AnyBone*, o sequenciador é uma função virtualizada de rede denominada de *VNF-Sequencer*. A *VNF-Sequencer* explora as funcionalidades da rede SDN, isto é, como há um controlador SDN responsável por gerenciar toda a comunicação da rede, a *VNF-Sequencer* recebe do controlador SDN todas as comunicações referentes as primitivas encaminhadas via *RBCast*. *RBCast* é uma biblioteca que possui a implementação das primitivas para a comunicação confiável, além disso ela oferece uma interface denominada de *IRBCast* (apresentada na Figura 8.3) para a comunicação com as aplicações.

No *AnyBone*, quando um processo deseja transmitir uma mensagem *m* por difusão confiável e atômica, a ordem total é construída através de 2 passos apresentados a seguir:

- Passo 1: primeiramente o emissor de uma mensagem *m* define o algoritmo que será utilizado na transmissão da mensagem, para isso ele acessa a primitiva de comunicação disponível pelo *AnyBone* e realiza uma difusão dentro da rede SDN. Esse processo é similar ao primeiro passo apresentado na Figura 8.1(b). Porém, essa difusão não terá sucesso, isto é, os demais processos da rede não recebem essa mensagem, somente a *VNF-Sequencer*. Esta estratégia é utilizada por dois motivos: em primeiro lugar a difusão permite tornar transparente o local da *VNF-Sequencer*; em segundo, a difusão é "abortada" para reduzir o número de mensagens. Na verdade, a difusão é cancelada por não deixar o controlador instalar a respectiva regra nos *switches*. Desta maneira, nossa estratégia utiliza um sequenciador que gera o mesmo número de mensagens do método UB, mas com as vantagens do método BB. Alternativamente, pode ser utilizada uma comunicação *unicast* que faz a conexão entre o processo emissor com o sequenciador, de maneira explicita. Por fim, antes da mensagem ser transmitida, *m* recebe um valor de sequência local atribuído pelo próprio emissor.

- Passo 2: quando a mensagem *m* é recebida pela *VNF-Sequencer*, o sequenciador ativa o algoritmo para difusão definido no passo 1, executando todos os procedimentos deste algoritmo. Quando *m* for retransmitida pela *VNF-Sequencer* aos processos receptores, a mensagem recebe um valor global de sequência atribuído pelo próprio sequenciador. Por fim, os processos receptores entregam *m* de acordo com o valor de sequência atribuído pela função de rede.

Note que a *VNF-Sequencer* precisa considerar duas sequências de mensagens, uma que respeita a ordem local de transmissão de cada processo, isso possibilita, por exemplo, que a ordem FIFO seja implementada, e uma ordem global que possibilita garantir que a ordem de entrega será atômica. Na transmissão atômica, a ordem global é sempre a ordem em que os processos finais (receptores) consideram para a entrega das mensagens.

Algorithm 5: Algoritmo para a construção da ordem global.

```
1  Sender
2    Init:
3      local_seq := 1;                          ▹ Contador de mensagem local
4      RBtype := (AtomicRB | AtomicFIFO | AtomicCAUSAL ); ▹ Define o RB
       algoritmo utilizado
5    forall p_i ∈ Π sends m
6      run broadcast (DATA, local_seq, RBtype) to sequencer;
7      local_seq := local_seq + 1;
8  VNF-Sequencer
9    Init:
10     global_seq := 1;   ▹ Contador de mensagem global utilizado para entregar m
11   when receive (m) do
12     run forward (DATA, local_seq, RBtype); ▹ Ativa o apropriado RB algoritmo
13   when send (m) do
14     run R_broadcast (DATA, global_seq, RBtype) ;
15     global_seq := global_seq + 1;
16 Receiver
17   Init:
18     nextMsg := global_seq;
19     pendingMsg := 0;
20   forall p_j ∈ Π receives m
21     pendingMsg := pendingMsg ∪ {m};
22     while (∃(m′ ∈ pendingMsg ∧ m′.global_seq = nextMsg ));
23       run deliver (m′);
24       pendingMsg := pendingMsg ∪ \ {m′};
25       nextMsg := nextMsg + 1;
```

No Algoritmo 5 é apresentado o pseudo código para a construção da ordem atômica entre os processos. No algoritmo há 3 papéis distintos: *Sender* (processo emissor), *VNF-Sequencer* (sequenciador das mensagens) e o *Receiver* (processo receptor). Para todo o processo p_i (*Sender*) que deseja transmitir uma mensagem m, por difusão atômica e confiável, inicialmente define o algoritmo a ser utilizado, por exemplo, difusão atômica, difusão atômica FIFO ou difusão atômica causal. A mensagem m será transmitida por difusão carregando informações como o contador local de mensagens ($local\_seq$) e informando o tipo de algoritmo que será utilizado.

A *VNF-Sequencer*, quando recebe m, repassa ($forward$) a mensagem para o algoritmo definido pelo emissor. Neste momento, o algoritmo que será ativado precisa levar em conside-

ração a informação do contador local encaminhado no cabeçalho da mensagem pelo emissor. Este é o caso que ocorre quando o Algoritmo atômico FIFO é ativado. Entretanto, para a difusão atômica basta respeitar a ordem global indicada por $global\_seq$ na *VNF-Sequencer*.

O contador global é inserido na mensagem e incrementado sempre que a *VNF-Sequencer* retransmite uma mensagem aos processos finais. O receptor (*Receiver*) ao receber a mensagem adiciona m na lista de mensagens pendentes ($pendingMsg$). Logo após ele verifica se há alguma mensagem m' que possui o contador igual ao identificador da próxima mensagem ($nextMsg$) e verifica também se m' ainda está em $pendingMsg$. Quando estas duas condições são contempladas, então m' e todas as outras mensagens que satisfazem as condições indicadas são entregues para a aplicação.

No caso da aplicação requerer ordenação FIFO, a *VNF-Sequencer* implementa a ordem de entrega de acordo com o Algoritmo 6. Neste algoritmo, o processo emissor envia uma mensagem por difusão para a *VNF-Sequencer* indicando o algoritmo a ser executado. A *VNF-Sequencer* executa o encaminhamento da mensagem para o algoritmo FIFO. No algoritmo FIFO é necessário verificar se m, transmitida por p_i, possui o valor do contador esperado como valor da próxima mensagem. Caso contrário, haverá uma pendência na ordem FIFO que precisa ser, antes de mais nada, resolvida. Neste caso, m é inserida na lista de pendências ($F\_pendingMsg$). Em outras palavras, se p_i transmitir $m_3^{p_i}$ e $m_4^{p_i}$ (onde $m_3^{p_i}$ é uma mensagem transmitida por p_i e 3 é o valor de local_seq) e a *VNF-Sequencer* receber primeiro m_4, ela armazena e atrasa a transmissão de m_4 até que m_3 seja recebida.

Quando a respectiva mensagem esperada (m_3) for recebida pelo algoritmo FIFO, ela e as mensagens que estão pendentes serão encaminhadas para a *VNF-Sequencer*. A ordem de encaminhamento das mensagens pendentes respeita a ordem do contador local inseridas em $F\_pendingMsg$. Então, *VNF-Sequencer* adiciona o valor do contador global à mensagem e transmite $m^s_{m'}$ e $m^s_{m''}$ por difusão confiável aos destinatários, onde $m' < m''$, m'=m_3 e m''=m_4. Por fim, os receptores entregam as mensagens de acordo com o Algoritmo 5 (*Receiver*), isto é, enquanto houver mensagens pendentes (mensagens armazenadas em $pendingMsg$) e com o valor do contador global igual ao respectivo valor esperado ($nextMsg$), então as mensagens são entregues nesta ordem para a aplicação.

Observe que neste exemplo se um outro processo p_j transmitir $m_1^{p_j}$ (primeira mensagem de p_j), a *VNF-Sequencer* ao receber esta mensagem a difunde imediatamente, pois $m_1^{p_j}$ é independente das mensagens transmitidas por p_i.

Conforme descrito em [Hadzilacos e Toueg, 1994], a difusão causal pode ser derivada de outras propriedades de algoritmos. No *AnyBone* a difusão atômica causal é implementada por satisfazer a ordem atômica FIFO e a ordem local (ver Seção 3.3.2). Neste caso, ao ativar o algoritmo para difusão atômica FIFO, automaticamente é garantida a ordenação atômica causal.

Para melhor compreensão, vamos considerar o mesmo exemplo apresentado na Seção 3.3.2, ou seja, em uma rede de notícias se os usuários transmitem suas mensagens por difusão confiável e na ordem atômica causal, o seguinte cenário deve ser respeitado: usuário A transmite sua notícia m_1^A (mensagem de A com o contador local 1). Outro usuário B entrega a notícia de A e responde com a notícia m_1^B (mensagem de B com o contador local 1) que só pode ser entendida por usuários que tenham visualizado a notícia transmitida por A (m_1^A). Neste caso, a ordem causal exige que um outro usuário qualquer C, entregue antes a mensagem transmitida por A e depois a mensagem transmitida de B; assim a resposta de B será bem interpretada.

Vamos agora, por contradição, imaginar que o usuário C entrega primeiro a mensagem transmitida por B e depois a mensagem transmitida por A, violando a ordem atômica causal. Usando o Algoritmo 6, este cenário é impossível de ocorrer. Pois se B transmite uma mensagem que possui dependência causal com a mensagem transmitida por A, ou seja, um evento que

Algorithm 6: Algoritmo para a construção da ordem FIFO.

```
1  Sender
2    Init:
3      RBtype := (AtomicFIFO);
4      local_seq := 1;
5    forall p_i ∈ Π F_broadcasts m
6      run R_broadcast (DATA, local_seq, RBtype) ;
7  VNF-Sequencer
8    Init:
9      nextMsg := 1;
10     F_pendingMsg := 0;                    ▹ Lista de mensagens pendentes
11 ...
12   when RBtype = AtomicFIFO do
13     if ( m.local_seq = nextMsg) then
14           run sequencer.send(m);
15           nextMsg := nextMsg + 1;
16           while (∃(m' ∈ F_pendingMsg ∧ m'.local_seq = nextMsg ));
17             run sequencer.send (m');
18             nextMsg := nextMsg + 1;
19             F_pendingMsg := F_pendingMsg ∪ \ {m'};
20     else
21           F_pendingMsg := F_pendingMsg ∪ {m'};
```

ocorreu em B posterior ao evento ocorrido em A, significa que o *sequencer* recebeu primeiro a mensagem de A e adicionou a esta mensagem o contador global c (mensagem de A com o contador global: m_c^A). Então, a mensagem de A é encaminhada para os outros processos por difusão confiável. Neste momento, para que B possa responder a mensagem transmitida de A, o usuário B precisa antes ter recebido m_c^A. Na sequência, B responde com outra mensagem. Novamente, o *sequencer* recebe a mensagem transmitida por B (a resposta da mensagem enviada por A) e adiciona o contador global c' (mensagem de B com o contador global: $m_{c'}^B$), onde $c < c'$. Se cada processo respeitar a ordem implementada pelo $receiver$ do Algoritmo 5. Logo, o usuário C não entrega $m_{c'}^B$ antes de entregar m_c^A, pois $m_c^A < m_{c'}^B$.

Observe que se as duas mensagens do exemplo anterior forem transmitidas pelo mesmo processo, a ordem será garantida pela propriedade do algoritmo FIFO através da análise do contador local realizada pelo *sequencer*. Note também que a abordagem modular utilizada para a construção do *AnyBone* facilita a implementação e a garantia das propriedades dos algoritmos. Por exemplo, o algoritmo FIFO necessita da difusão confiável, a ordem causal é dependente do algoritmo FIFO e todos dependem de um sequenciador para que a ordem atômica seja garantida. Essa abordagem reduz a complexidade de implementação dos algoritmos haja vista que suas funcionalidades estão compartilhadas com outros algoritmos.

8.2.3 A Arquitetura e a Implementação do AnyBone

A arquitetura do *AnyBone* é composta por componentes que estão espalhados dentro da rede SDN. Para disponibilizar as primitivas de comunicação às aplicações, o *AnyBone* oferece uma interface na biblioteca denominada de *RBCast*. A arquitetura pode ser visualizada na Figura 8.2. Observe que a *RBCast* está, estrategicamente, nas pontas da comunicação, isto é, nos processos emissores e receptores. *RBCast* se comunica com a *VNF-Sequencer* através de um controlador SDN que repassa pacotes para um outro componente denominado de *Filter* (ver Capítulo 7).

A *Filter* é composta por dois módulos denominados de *Classifier* e *Forwarder*. O *Classifier* possui acesso às regras que são armazenadas no repositório *Rules*. Essas regras permitem classificar os fluxos de entrada para determinar o destino de cada pacote. O processo ocorre da seguinte forma: o pacote (*packet-in*) chega no *switch* OpenFlow que é encaminhado ao controlador SDN (passo 1). O controlador encaminha todos os pacotes para a ferramenta *Filter* (passo 2), se houver alguma regra cadastrada no módulo *Classifier* que se encaixe ao pacote recebido (passo 3), o pacote é repassado para o módulo *Forwarder* (passo 5), caso contrário o pacote será retornado ao controlador (passo 4). O *Forwarder*, por sua vez, é responsável por realizar a conexão com repasse dos pacotes para a VNF que pode estar localizada em qualquer ponto da rede (passo 6). Esse processo também é identificado de SFP (*Service Function Path*). Finalmente a *VNF-Sequencer* recebe o pacote e inicializa a difusão na rede conforme o algoritmo definido pela aplicação (passo 7).

Note que a *VNF-Sequencer* está sendo executada em um *container* no ambiente Docker[2]. Resumidamente, a implementação em *containers* é motivada por sua baixa utilização dos recursos computacionais, rápida instanciação e fácil implementação de funções virtualizadas, como foi mostrado nos experimentos do Capítulo 5.

Os algoritmos no *AnyBone* estão implementados e estruturados conforme o diagrama de classes apresentado na Figura 8.3. Por simplicidade, não listamos no diagrama os atributos e os métodos implementados em cada classe, somente as associações que permitem compreender a lógica implementada pelo *AnyBone*. A Figura 8.3 está dividida em três conjuntos de classes:

[2]http://www.docker.org/

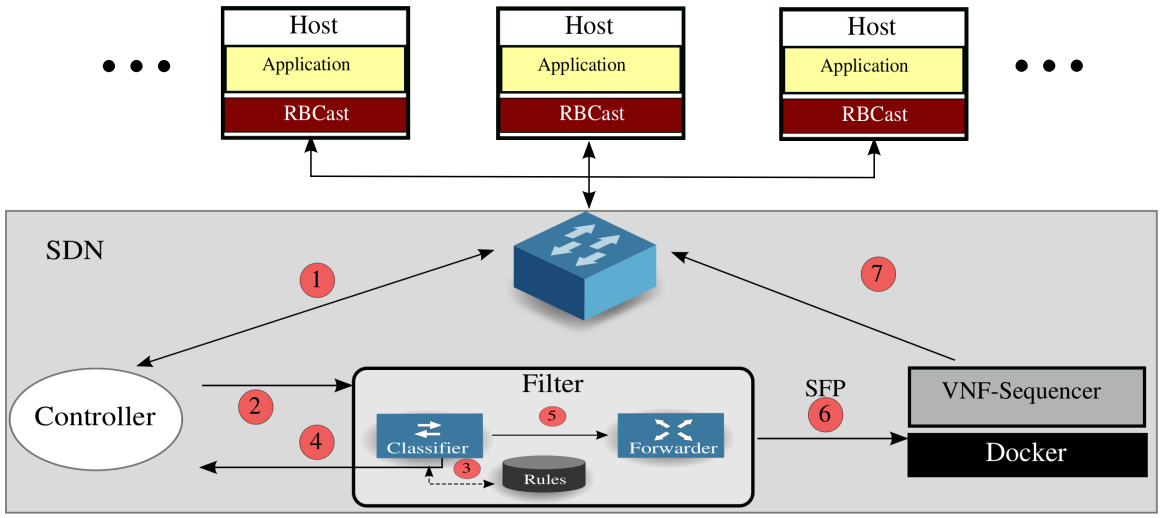


Figura 8.2: Arquitetura do AnyBone

RBCast, *VNF-Sequencer* e o *Repositório de Algoritmos*. *RBCast* executa nos processos finais como apresentado na arquitetura, *VNF-Sequencer* é o sequenciador que é uma função de rede que executa em conjunto com o controlador SDN e o *Repositório de Algoritmos* que faz parte tanto da *VNF-Sequencer* quanto do *RBCast* e que permite a inclusão dos algoritmos em uma estrutura modular.

Uma aplicação (`APP`) usa o `RBManager` para especificar o algoritmo de difusão que será utilizado e para encaminhar as mensagens aos destinatários. Além disso, a `APP` utiliza a interface `IRBCast` para receber as mensagens que serão entregues por *callback* para a aplicação. Neste caso, a aplicação precisa fazer um registro prévio no *AnyBone* indicando sua instância através do método abstrato declarado na interface `IRBCast`.

O `RBManager` instancia o algoritmo indicado pela `APP` através da interface `RB`. A interface `RB` comunica todos os algoritmos e especifica funções comuns que são implementadas por eles, como exemplo, oferecer as primitivas de *unicast* e *broadcast*, que podem ser utilizadas para comunicar com o sequenciador e com os processos finais, ou ainda inicializar uma *thread* (`Listener`) para a troca de mensagens. As mensagens são estruturadas de acordo com a classe `Message`, isto é, possui 4 parâmetros: conteúdo, identificador do processo, sequência da mensagem e o tipo. A mensagem pode ser de 3 tipos: $msgBC$ (mensagem para ser entregue por difusão), $msgRequestSequencer$ (requisição do endereço IP do sequenciador) e $msgReplySequencer$ (mensagem de resposta do sequenciador).

A classe `Lib`, não listada no diagrama por não apresentar relações entre as classes, define valores que são estáticos, como exemplo, tipo de cada algoritmo, tipo da mensagem, portas utilizadas para a comunicação e tamanho das mensagens.

Os algoritmos são estruturados de maneira modular no *Repositório de Algoritmos*. O `ReliableBroadcast` é a base em que todos os demais algoritmos herdam funcionalidades como, por exemplo, a utilização da primitiva para a difusão confiável. Além disso, o `ReliableBroadcast`, quando necessário, repassa as mensagens recebidas para seus respectivos algoritmos de ordenação de mensagens. O algoritmo `ReliableBroadcast` executa a entrega de uma mensagem (*deliver(m)*) somente se a `APP` habilitou o serviço de difusão confiável. Por outro lado, se for habilitado outro algoritmo, por exemplo, difusão confiável FIFO, então a entrega será realizada pelo algoritmo `FifoBroadcast`.

Para a difusão atômica a entrega ocorre praticamente da mesma maneira, porém com o fluxo sendo direcionado para a classe `SequencerListener`. Ou seja, a mensagem é recebida

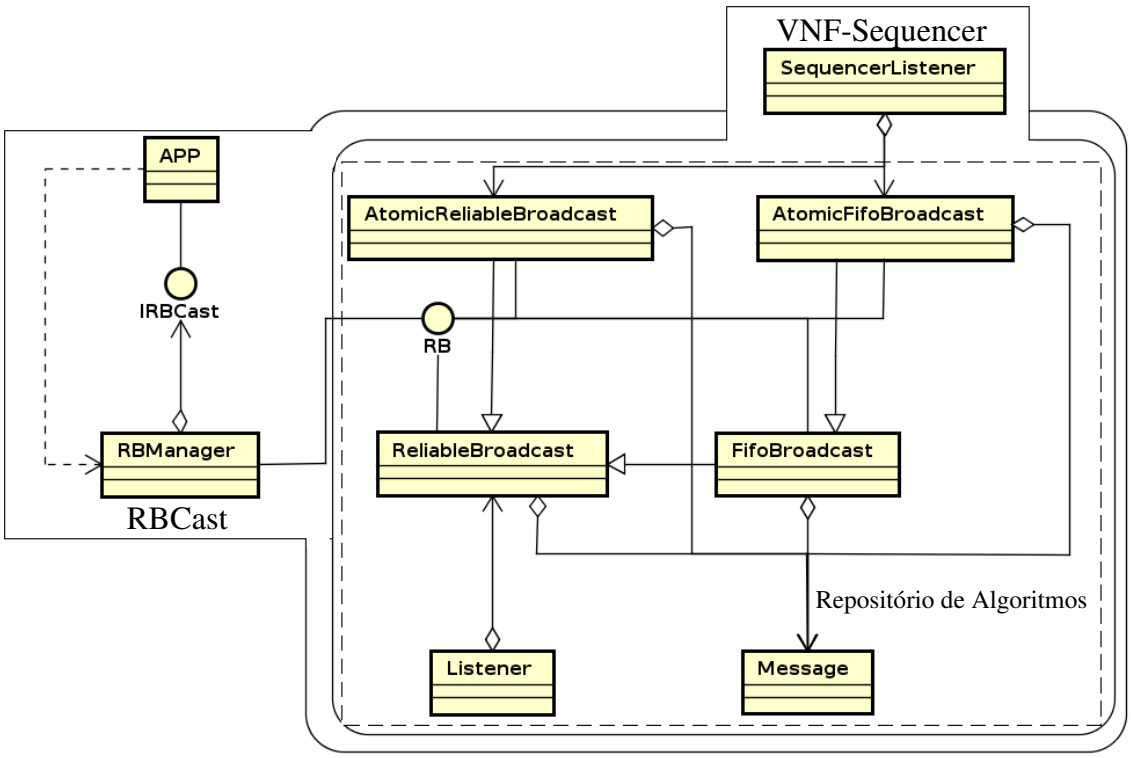


Figura 8.3: Diagrama de classes do *AnyBone*.

pelo `SequencerListener` que habilita o algoritmo para a difusão atômica. A comunicação continua sendo realizada via primitiva de *broadcast* utilizando a classe `ReliableBroadcast` com o contador global inserido pelo sequenciador na mensagem. A entrega da mensagem (*deliver(m)*) ocorre de acordo com o contador global definido pelo sequenciador. Como explicado na seção anterior, o encaminhamento de uma mensagem *m* com o contador global só é realizado quando não forem detectadas pendências pelo algoritmo em execução. Por fim, não havendo pendências, *m* será encaminhada aos processos finais que irão entregar *m* conforme o contador global e de acordo com o *Receiver* apresentado no Algoritmo 5.

Tratamento de Falhas

Note que até o momento não temos considerado a hipótese da ocorrência de falhas durante a execução do *AnyBone*. Entretanto, o *AnyBone* trabalha com esta hipótese. Em especial, ele trata falhas que ocorrem em processos que param completamente de executar suas funções de maneira prematura. Este tipo de falha é denominado de *crash* (colapso). Na presença de falhas, *AnyBone* faz uso de um detector de falhas responsável por monitorar os processos. O detector de falhas utilizado é denominado de NFV-FD proposto no Capítulo 5.

No Capítulo 5 também foi apresentado o Algoritmo 2 para a difusão confiável das mensagens usando um detector de falhas. No Algoritmo 2 há duas situações que podem forçar um processo a retransmitir uma mensagem: (*i*) ocorre quando um processo percebe a falha do emissor antes de entregar *m*, e (*ii*) acontece sempre que um processo entrega *m* e percebe que o seu emissor falhou. Em outras palavras, se o emissor da mensagem não falhar ao transmitir uma mensagem por difusão confiável, os receptores não retransmitem *m* novamente.

O *AnyBone* é formado por um sequenciador que nunca falha. Além disso, considera-se um canal confiável, isto é, não cria, não perde e não altera mensagens transmitidas. Na próxima seção são descritos os resultados experimentais onde também é avaliado o funcionamento do *AnyBone* em diversos cenários de execução.

8.3 Avaliação Experimental do AnyBone

Nesta seção, são executados experimentos com o objetivo de avaliar o desempenho do *Anybone* em uma rede SDN. O ambiente para execução dos experimentos é implementado usando o protocolo *OpenFlow* e controladores *Ryu*[3]. O ambiente é hospedado em uma máquina física com as seguintes características: processador AMD FX-4300 CPU 3.8GHz com 4 núcleos e sistema operacional *Ubuntu* 16.04 com *kernel* 4.4.0-53.

Os *switches* SDN foram executados através da ferramenta *cbench* detalhada na Seção 7.4. A *VNF-sequencer* é hospedada em um *container* no ambiente Docker, bem como o próprio controlador Ryu. A seguir são apresentados 2 grupos de experimentos: (1) análise do custo da utilização do sequenciador e (2) analise do *throughput* do AnyBone.

Analise do custo da utilização do sequenciador

Na seção anterior, vimos que o *AnyBone* garante a entrega atômica por utilizar um sequenciador implementado dentro da rede SDN. Esta abordagem é atraente pois facilita o gerenciamento das mensagens pelo sequenciador que recebe toda a comunicação da difusão confiável. Além da entrega atômica, o sequenciador gerencia a ordem em que as mensagens são entregues às aplicações. Por outro lado, como o sequenciador centraliza toda a comunicação entre os processos, há um custo a ser pago pela utilização desta abordagem. Neste sentido, o presente experimento tem por objetivo medir tal impacto comparando uma comunicação de difusão confiável e atômica (com o uso de um sequenciador) com uma comunicação que somente garante a difusão confiável (sem o uso de um sequenciador). A comparação é realizada em termos da latência de entrega das mensagens como explicado na sequência.

No experimento da Figura 8.4 foi utilizada uma topologia com um *switch* SDN, um controlador e variado número de processos que participam da difusão confiável. Um dos processos participantes do experimento implementa uma aplicação cliente responsável por inicializar o envio das mensagens que são, inicialmente, repassadas ao *RBCast* local. O *RBCast* realiza uma difusão confiável, no caso em que a comunicação não é atômica, ou o fluxo é desviado para a *VNF-Sequencer*, no caso da transmissão ser atômica. Quando o *RBCast* recebe o pacote, a mensagem é então entregue para a aplicação.

Neste experimento medimos a latência para a entrega das mensagens que corresponde ao tempo em que uma mensagem foi transmitida por difusão confiável até o instante em que o último processo tenha entregado a mensagem para a aplicação. Para cada experimento são executados 50 *broadcasts*. Os dados apresentados são valores médios de 3 amostras onde o intervalo de confiança utilizado é de 95%.

No gráfico da Figura 8.4, o eixo *Y* apresenta o valor da latência em milissegundos para a entrega das mensagens para as aplicações, ao passo que o eixo *X* descreve o número de processos participantes da difusão confiável. A comunicação ocorre com sequenciador (curva 'Com sequenciador') e sem sequenciador (curva 'Sem sequenciador'). Como previsto, é possível notar que a latência é maior com o uso do sequenciador, o que desejamos aferir são os valores

[3]https://osrg.github.io/ryu/

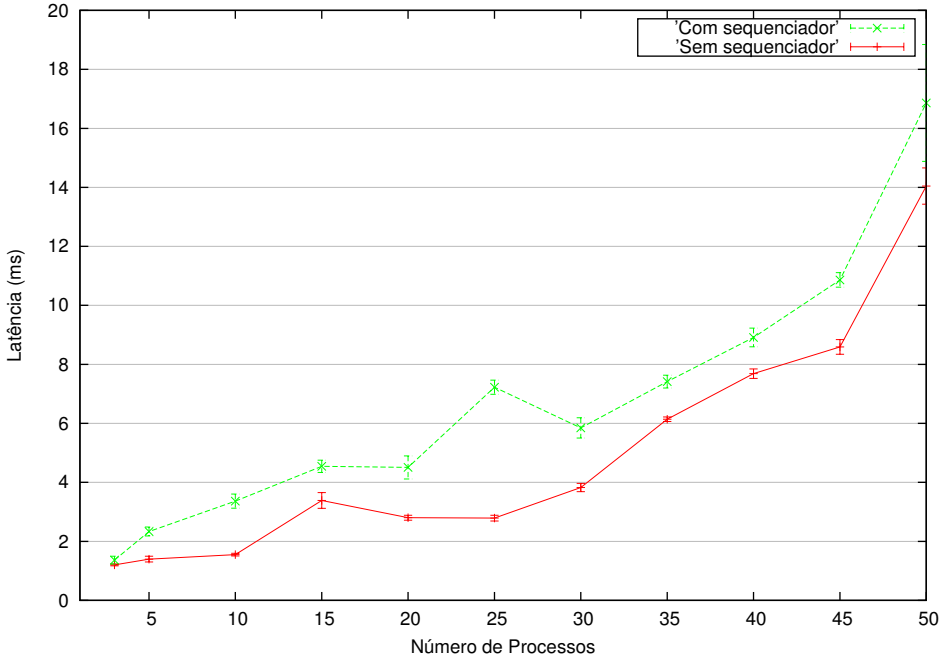


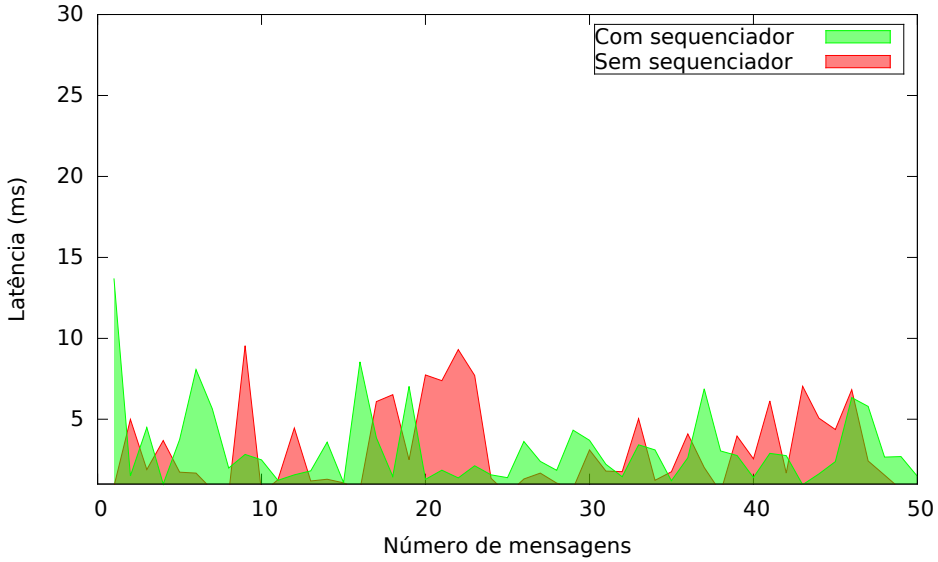
Figura 8.4: Comparação da latência para a entrega das mensagens, com e sem sequenciador.

exatos deste custo. Neste caso, observando os valores da latência, para todas as execuções, obteve-se aproximadamente 30,22% de acréscimo total. É possível notar também que a latência cresce para ambos os casos conforme o número de processos aumenta. Entretanto, para algumas amostrar observa-se que o crescimento não segue um valor padrão, é o caso que ocorre com 25 processos. Mesmo os valores sendo dados médios de 3 execuções, percebe-se algumas variações maiores na comunicação, o que eleva a média.

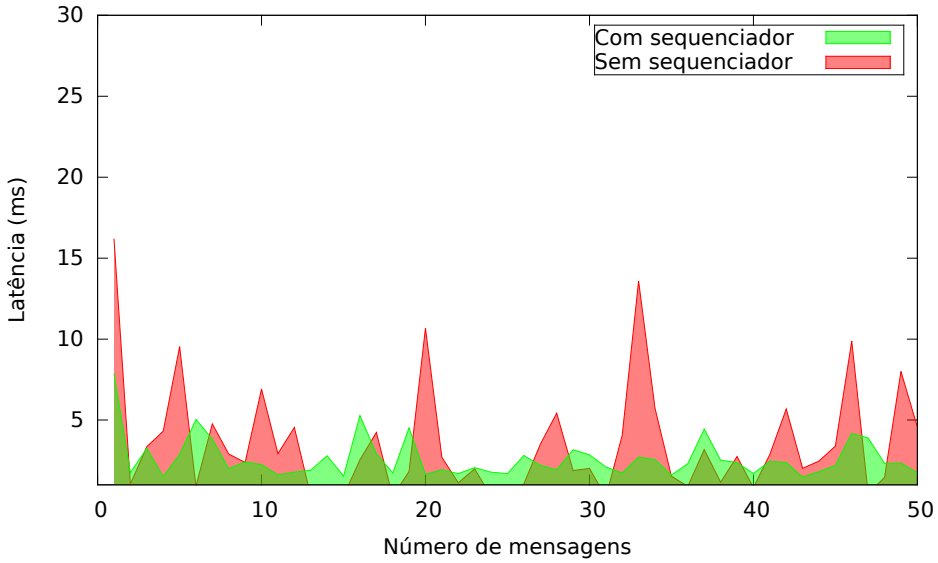
Outra análise que pode ser observada é a comparação de crescimento da latência de 5 processos para 50 processos. Neste caso, com sequenciador o aumento foi de aproximadamente 7 vezes, ao passo que sem sequenciador o aumento foi de 10 vezes. Portanto, pode-se dizer que a tendência é que a diferença da latência, pelo uso do sequenciador, seja atenuada conforme o número de processos cresce. Por outro lado, o sequenciador demonstra ser mais susceptível ao tamanho do pacote. No experimento da Figura 8.5 é calculada a latência de comunicação com 10 processos onde o tamanho do pacote aumenta de 128 bytes (Figura 8.5(a)) para 1024 bytes (Figura 8.5(b)). Os dados apresentados são valores médios de 3 amostras onde são executadas 50 transmissões de *broadcast*.

Com pacotes de 128 bytes observa-se que os valores são próximos para os dois casos analisados, ou seja, sem sequenciador a latência média obtida é de 3,07ms, ao passo que com o uso do sequenciador a latência média é de 3,27. Entretanto, ao analisarmos a Figura 8.5(b) onde o tamanho do pacote é aumentado para 1024 bytes, é possível observar grandes picos na comunicação com o uso do sequenciador. Esses valores elevam a média da latência, ou seja, sem sequenciador a latência média obtida é de 3,51ms e com o uso do sequenciador a latência média é de 5,94.

Diante das análises comparativas entre o uso ou não de um sequenciador, vale observar que a comunicação realizada com o sequenciador é atômica, ao passo que, sem o sequenciador a comunicação é somente uma difusão confiável não garantindo que todos os processos irão receber as mensagens na mesma ordem. Portanto, conclui-se que os benefícios proporcionados pelo uso do sequenciador compensam a latência obtida nesta abordagem.



(a) Latência com pacotes de 128 bytes.



(b) Latência com pacotes de 1024 bytes.

Figura 8.5: Comparação da latência com diferentes tamanhos de pacotes.

Análise do *throughput* no AnyBone

O próximo experimento tem por objetivo verificar o *throughput* do sistema variando o número de transmissores, controladores e *switches*, proporcionalmente. No experimento, cada execução tem duração de 3 minutos, são apresentados dados médios de 3 execuções e então o número de transmissores, controladores e *switches* é aumentado respectivamente, e os testes são novamente executados. É computado o número de pacotes processados por segundo no sequenciador. Dessa forma é calculado, para um intervalo de tempo, quantos pacotes foram executados e repassados para os destinatários que totalizam 10 processos. Cada transmissor acessa um *switch* que é gerenciado por um controlador SDN. Este cenário demonstra que o *AnyBone* pode ser executado com múltiplos controladores SDN.

É possível notar na Figura 8.6 que o *throughput* cresce quase linearmente com o número de transmissores, isto é, conforme o número de transmissores aumenta na rede, o número de pacotes transmitidos aumenta, fazendo o sequenciador lidar com um grande número de pacotes. Isso pode ser confirmado com os valores atingidos do *throughput* apresentados no eixo Y. Em especial, para até 10 transmissores o sequenciador não demonstrou ser um gargalo, atingindo um processamento de aproximadamente 900 pacotes por segundo.

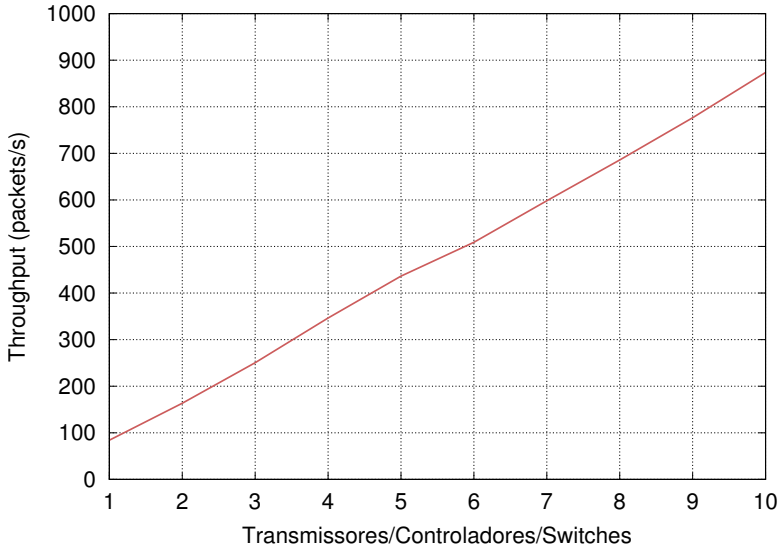


Figura 8.6: Throughput para a difusão atômica aumentando o número de transmissores/controladores/*switches*.

Ciente de que haverá um limite de processamento do sequenciador, onde ele poderá representar um gargalo para a comunicação do *AnyBone*, uma alternativa é distribuir a carga em sequenciadores móveis [Défago et al., 2004] ou, até mesmo, implementar sua lógica dentro de dispositivos dedicados como em *switches* proposto em [Li et al., 2016]. Entretanto, *AnyBone* encontra-se em fase de construção onde diversos assuntos ainda serão tratados. Por fim, acredita-se que diante da análise prévia apresentadas neste capítulo, visionamos a possibilidade real da própria rede realizar a entrega atômica e confiável de mensagens, liberando as aplicações de tal tarefa.

8.4 Considerações Parciais

Neste capítulo propomos *AnyBone*, uma função virtualizada de rede que oferece as primitivas de difusão confiável para garantir a entrega ordenada das mensagens transmitidas na rede. Para realizar tal função, *AnyBone* divide a responsabilidade entre dois componentes: *VNF-Sequencer* que é um sequenciador que gerencia as transmissões e entrega as mensagens ordenadas aos processos; e *RBCast* que oferece uma API para as aplicações trocarem mensagens utilizando as primitivas de difusão confiável.

Nos resultados experimentais mostramos que o *AnyBone* consegue garantir a ordem atômica das mensagens pelo uso de um sequenciador, essa abordagem facilita a implementação do serviço. Por outro lado, há um custo a ser pago em relação a latência para a entrega das mensagens. O custo da latência foi apresentado em diferentes cenários, ou seja, variando o número de participantes da difusão e o tamanho das mensagens transmitidas. Por fim, foi medido o *throughput* aumentando o número de transmissores, *switches* e controladores SDN. No

experimento observou-se que para até 10 transmissores o sequenciador não demonstrou ser um gargalo.

Capítulo 9

Conclusão

No presente trabalho foi proposta a implementação de serviços de tolerância a falhas que são utilizados por aplicações distribuídas e por múltiplos controladores auxiliando na sincronização consistente da rede SDN. Estes serviços garantem as operações dos processos mesmo na presença de falhas. Inicialmente, foi proposto um serviço para a detecção de falhas adaptativo, ou seja, o serviço utiliza uma estratégia que procura estimar o intervalo do *timeout* de acordo com atrasos na comunicação. O serviço proposto foi inserido e adaptado em dois diferentes contextos, assuntos exibidos a seguir.

Na primeira contribuição foi proposta um serviço para detecção de falhas em processos executando em diferentes sistemas autônomos na Internet. O serviço denominado de IFDS provê diversas funcionalidades implementadas através de uma arquitetura composta por agentes SNMP, *Web Services* e uma MIB denominada de $fdMIB$. A $fdMIB$ executa as ações de monitoramento dos estados dos processos com referência nos parâmetros de QoS que são fornecidos pelas aplicações. Para garantir estes parâmetros de QoS, foram propostas duas estratégias (η_{max} e η_{GCD}) que permitem a configuração do IFDS tendo em vista os requisitos de múltiplas aplicações.

Os resultados experimentais apresentados mostraram que, a estratégia η_{max} é mais apropriada para aplicações que toleram maiores atrasos na comunicação (caso da Internet), ao passo que a estratégia η_{GCD} é indicada para uso em redes locais, obtendo maior probabilidade para uma resposta exata e menor tempo para a detecção de uma falha. Vimos também que quando os parâmetros de QoS não são violados, eventuais falsas detecções cometidas pelo detector de falhas, podem ser omitidas para as aplicações. Foi mostrada ainda a eficiência da estratégia utilizada para a adaptação do *timeout* em diferentes situações de atrasos de comunicação. Por fim, mesmo considerando os atrasos adicionados pelo uso de *Web Services*, os experimentos executados no PlanetLab demostraram a viabilidade de usar o IFDS para as comunicações entre diferentes sistemas autônomos na Internet.

No segundo contexto, foi proposta uma NFV para detecção de falhas de processos e enlaces de comunicação em uma rede OpenFlow. A NFV proposta foi denominada de NFV-FD. O trabalho apresentou e detalhou questões sobre a implementação da NFV. Em especial, implementamos a NFV-FD de diferentes maneiras: dentro e fora do controlador SDN, utilizando máquinas virtuais tradicionais e hospedada em *containers*. Os experimentos mostraram que a implementação de uma NFV junto ao controlador causa um impacto relevante no uso do processador sendo, portanto, uma boa estratégia de projeto concentrá-la fora do controlador, mesmo com o ligeiro aumento no tempo de detecção de falhas demonstrado nos experimentos. Além disso, os resultados que comparam a implementação da NFV-FD em máquinas virtuais e

em *containers*, mostraram que *containers* são uma tecnologia atrativa por ser leve, em especial quanto à baixa utilização dos recursos e rápida instanciação dos serviços.

Com base nos estudos para o cômputo do *timeout*, nos trabalhos sobre detector de falhas, surgiu outra proposta onde apresentamos uma estratégia para beneficiar o mecanismo de detecção de falhas, tendo como foco melhorar o tempo de detecção e o número de falsas suspeitas cometidas pelo detector. A estratégia denominada de $tuning_{\phi}$ reajusta o valor do *timeout* de acordo com os tempos de comunicação calculados, de maneira a refletir o comportamento real da rede. Na prática, adaptamos o cálculo proposto por Jacobson tornando o *timeout* mais fiel ao comportamento do ambiente. Nos experimentos realizados constatou-se que $tuning_{\phi}$ reduz de forma expressiva o número de falsas suspeitas, mesmo em cenários onde os atrasos são maiores, como é o caso da Internet. Além disso, $tuning_{\phi}$ apresenta uma redução no tempo de detecção de falhas mantendo um bom desempenho no tempo médio para correção de falsas suspeitas.

Um outro problema abordado tratou da sincronização consistente em redes SDN onde o plano de controle é distribuído. Neste cenário, há a necessidade de realizar a sincronização entre todos os controladores envolvidos. Porém, vimos que a sincronização entre os múltiplos controladores distribuídos não é uma tarefa trivial. Foi então proposta uma solução para a sincronização do plano de controle das redes SDN, na qual um grupo de controladores distribuídos se mantêm consistente com o auxílio de uma função virtualizada de rede denominada de *VNF-Consensus*. A *VNF-Consensus* implementa o algoritmo de consenso Paxos no ambiente de rede. Dessa forma, todas as decisões realizadas pela *VNF-Consensus* são executadas sem a atuação direta dos controladores. Os resultados experimentais mostraram que quando os próprios controladores executam a sincronização do plano de controle, há um impacto negativo no desempenho global da rede. Por outro lado, com a utilização da *VNF-Consensus*, o plano de controle é sincronizado sem aumentar a carga de trabalho nos controladores. Por consequência, vimos que há melhorias no desempenho da rede. A estratégia se diferenciou das demais encontradas na literatura por não estar implementada dentro dos *switches* e, nem sequer, nos controladores SDN.

Por fim, propomos *AnyBone*, uma função virtualizada de rede que oferece as primitivas de difusão confiável para garantir a entrega ordenada das mensagens transmitidas na rede. Para realizar tal função, *AnyBone* divide a responsabilidade entre dois componentes: *VNF-Sequencer* que é um sequenciador que gerencia as transmissões e entrega as mensagens ordenadas aos processos; e *RBCast* que oferece uma API para as aplicações trocarem mensagens utilizando as primitivas de difusão confiável.

Nos resultados experimentais mostramos que o *AnyBone* consegue garantir a ordem atômica das mensagens pelo uso de um sequenciador, essa abordagem facilita a implementação do serviço. Por outro lado, há um custo a ser pago em relação a latência para a entrega das mensagens. O custo da latência foi apresentado em diferentes cenários, ou seja, variando o número de participantes da difusão e o tamanho das mensagens transmitidas. Diante desta análise comparativa, vale observar que a comunicação realizada com o sequenciador é atômica, ao passo que, sem o sequenciador a comunicação é somente uma difusão confiável não garantindo que todos os processos irão receber as mensagens na mesma ordem. Portanto, conclui-se que os benefícios proporcionados pelo uso do sequenciador compensam a latência obtida nesta abordagem. Por último, foi medido o *throughput* aumentando o número de transmissores, *switches* e controladores SDN. No experimento observou-se que para até 10 transmissores o sequenciador não demonstrou ser um gargalo.

Esta tese demonstrou não apenas a viabilidade, mas também os benefícios de implementar serviços distribuídos confiáveis na própria rede utilizando técnicas de virtualização como NFVs para serem disponibilizadas como serviços para aplicações distribuídas. Por fim, trabalhos futuros incluem, no contexto da última contribuição desta tese, tolerar falhas no sequenciador,

componente responsável por gerenciar as mensagens entregues no *AnyBone*; distribuir os sequenciadores na rede com o objetivo de dividir as tarefas entre eles provendo melhor latência e *throughput*; no contexto das aplicações permitir fácil integração do *AnyBone* com diversas linguagens de programação; manter um ambiente de desenvolvimento e de fácil acesso via *Docker Hub*. Destaca-se ainda estudar e implementar otimizações dos algoritmos para difusão confiável, como também, executar experimentos utilizando aplicações reais como assegurar a consistência de serviços replicados em data centers. No contexto da *VNF-Consensus*, planeja-se ainda a implementação de um serviço para a sincronização do plano de dados de uma rede SDN, isto é, visando manter a consistência entre todos os *switches* da rede.

Referências Bibliográficas

[IEE, 2009] (2009). IEEE standard for local and metropolitan area networks– station and media access control connectivity discovery. *IEEE Std 802.1AB-2009 (Revision of IEEE Std 802.1AB-2005)*.

[Anderson et al., 2016] Anderson, J., Hu, H., Agarwal, U., Lowery, C., Li, H. e Apon, A. (2016). Performance considerations of network functions virtualization using containers. Em *International Conference on Computing, Networking and Communications (ICNC'16)*.

[Batalle et al., 2013] Batalle, J., Ferrer Riera, J., Escalona, E. e Garcia-Espin, J. (2013). On the Implementation of NFV over an OpenFlow Infrastructure: Routing Function Virtualization. Em *Future Networks and Services (SDN4FNS)*.

[Berde et al., 2014] Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., Koide, T., Lantz, B., O'Connor, B., Radoslavov, P., Snow, W. e Parulkar, G. (2014). ONOS: Towards an Open, Distributed SDN OS. Em *3th Workshop on Hot Topics in Software Defined Networking (HotSDN)*.

[Bertier et al., 2003] Bertier, M., Marin, O. e Sens, P. (2003). Performance Analysis of a Hierarchical Failure Detector. Em *International Conference on Dependable Systems and Networks (DSN)*.

[Bondan et al., 2014] Bondan, L., Dos Santos, C. e Zambenedetti Granville, L. (2014). Management requirements for ClickOS-based Network Function Virtualization. Em *10th International Conference on Network and Service Management (CNSM)*.

[Bondan et al., 2017] Bondan, L., Wauters, T., Volckaert, B., Turck, F. D. e Zambenedetti Granville, L. (2017). Anomaly Detection Framework for SFC Integrity in NFV Environments. Em *3rd IEEE Conference on Network Softwarization (NetSoft 2017)*.

[Borran et al., 2012] Borran, F., Hutle, M., Santos, N. e Schiper, A. (2012). Quantitative analysis of consensus algorithms. *IEEE Transactions on Dependable and Secure Computing*, 9(2).

[Braden, 2017] Braden, R. (Acessado em 01/06/2017). Requirements for internet hosts - communication layers. `https://tools.ietf.org/html/rfc1122`.

[Buyya et al., 2014] Buyya, R., Calheiros, R. N., Son, J., Dastjerdi, A. V. e Yoon, Y. (2014). Software-Defined Cloud Computing: Architectural elements and open challenges. Em *International Conference on Advances in Computing, Communications and Informatics (ICACCI)*.

[Cachin et al., 2011] Cachin, C., Guerraoui, R. e Rodrigues, L. (2011). *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd edition.

[Canini et al., 2015] Canini, M., Kuznetsov, P., Levin, D. e Schmid, S. (2015). A distributed and robust SDN control plane for transactional network updates. Em *IEEE Conference on Computer Communications (INFOCOM)*.

[Chandra e Toueg, 1996] Chandra, T. D. e Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2).

[Chen et al., 2000] Chen, W., Toueg, S. e Aguilera, M. K. (2000). On the Quality of Service of Failure Detectors. Em *International Conference on Dependable Systems and Networks (DSN)*.

[Correia et al., 2006] Correia, M., Neves, N. F. e Veríssimo, P. (2006). From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82–96.

[Cotroneo et al., 2014] Cotroneo, D., De Simone, L., Iannillo, A., Lanzaro, A., Natella, R., Fan, J. e Ping, W. (2014). Network Function Virtualization: Challenges and Directions for Reliability Assurance. Em *International Symposium on Software Reliability Engineering Workshops (ISSREW)*.

[Cziva et al., 2015] Cziva, R., Jouet, S. e Pezaros, D. (2015). Container-based network function virtualization for software-defined networks. Em *International Symposium on Computers and Communications (ISCC'15)*.

[Dang et al., 2015] Dang, H. T., Sciascia, D., Canini, M., Pedone, F. e Soulé, R. (2015). Netpaxos: Consensus at network speed. Em *ACM Sigcomm Symposium on SDN Research (SOSR)*.

[de Camargo e Duarte, 2017] de Camargo, E. T. e Duarte, E. P. (2017). *Tolerância a Falhas em Sistemas MPI com Grupos Dinâmicos de Processos Recomendados e Registro de Mensagens Distribuído Baseado em Paxos*. Tese de doutorado, Universidade Federal do Paraná - UFPR, Curitiba - Brasil.

[de Sá e de Araújo Macêdo, 2010] de Sá, A. S. e de Araújo Macêdo, R. J. (2010). Qos self-configuring failure detectors for distributed systems. Em *International Conference on Distributed Applications and Interoperable Systems (DAIS)*. Springer-Verlag.

[Défago et al., 2004] Défago, X., Schiper, A. e Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421.

[Dixit e Casimiro, 2010] Dixit, M. e Casimiro, A. (2010). Adaptare-FD: A Dependability-Oriented Adaptive Failure Detector. *Symposium on Reliable Distributed Systems*.

[Ekwall e Schiper, 2007] Ekwall, R. e Schiper, A. (2007). Modeling and validating the performance of atomic broadcast algorithms in high latency networks. Em *13th International Euro-Par Conference*.

[ETSI, 2015a] ETSI (2015a). ETSI GS NFV 001: Network Functions Virtualisation (NFV); Use Cases. `http://www.etsi.org/deliver/etsi_gs/NFV/001_099/001/01.01.01_60/gs_NFV001v010101p.pdf`. Acessado em 22/05/2017.

[ETSI, 2015b] ETSI (2015b). ETSI gs nfv 002: Architectural Framework. http://www.etsi.org/technologies-clusters/technologies/nfv. Acessado em 22/05/2017.

[ETSI, 2015c] ETSI (2015c). Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges and Call for Action. http://portal.etsi.org/NFV/NFV_White_Paper.pdf. Acessado em 22/05/2017.

[ETSI, 2015d] ETSI (2015d). Network Functions Virtualisation (NFV); Management and Orchestration. http://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_nfv-man001v010101p.pdf. Acessado em 22/05/2017.

[ETSI, 2015e] ETSI (2015e). Network Functions Virtualisation (NFV); NFV Security; Privacy and Regulation; Report on Lawful Interception Implications. http://www.etsi.org/deliver/etsi_gs/NFV-SEC/001_099/004/01.01.01_60/gs_NFV-SEC004v010101p.pdf. Acessado em 14/06/2017.

[Eugster et al., 2004] Eugster, P. T., Guerraoui, R. e Kouznetsov, P. (2004). D-reliable broadcast: A probabilistic measure of broadcast reliability. Em *24th International Conference on Distributed Computing Systems (ICDCS)*.

[Felber et al., 1999] Felber, P., Defago, X., Guerraoui, R. e Oser, P. (1999). Failure detectors as first class objects. Em *International Symposium on Distributed Objects and Applications.*

[Felber et al., 1998] Felber, P., Guerraoui, R. e Schiper, A. (1998). The implementation of a CORBA object group service. *Theory and Practice of Object Systems*, 4(2).

[Ferrer Riera et al., 2014] Ferrer Riera, J., Escalona, E., Batalle, J., Grasa, E. e Garcia-Espin, J. (2014). Virtual Network Function Scheduling: Concept and Challenges. Em *International Conference on Smart Communications in Network Technologies (SaCoNeT)*.

[Fetzer, 2001] Fetzer, C. (2001). Enforcing Perfect Failure Detection. Em *21st International Conference on Distributed Computing Systems (ICDCS)*.

[Firoozjaei et al., 2017] Firoozjaei, M. D., Jeong, J. P., Ko, H. e Kim, H. (2017). Security challenges with network functions virtualization. *Future Generation Comp. Syst.*, 67.

[Fischer et al., 1985] Fischer, M. J., Lynch, N. A. e Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 32(2).

[Fukushima et al., 2014] Fukushima, M., Yoshida, Y., Tagami, A., Yamamoto, S. e Nakao, A. (2014). Toy Block Networking: Easily Deploying Diverse Network Functions in Programmable Networks. Em *38th International Computer Software and Applications Conference Workshops (COMPSACW)*.

[Guerraoui e Rodrigues, 2006] Guerraoui, R. e Rodrigues, L. (2006). *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Hadzilacos e Toueg, 1994] Hadzilacos, V. e Toueg, S. (1994). A modular approach to fault-tolerant broadcasts and related problems. Relatório técnico.

[Haleplidis et al., 2014] Haleplidis, E., Denazis, S., Koufopavlou, O., Lopez, D., Joachimpillai, D., Martin, J., Salim, J. e Pentikousis, K. (2014). ForCES Applicability to SDN-Enhanced NFV. Em *Third European Workshop on Software Defined Networks (EWSDN)*.

[Han et al., 2015] Han, B., Gopalakrishnan, V., Ji, L. e Lee, S. (2015). Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2).

[Hayashibara et al., 2004] Hayashibara, N., Défago, X., Yared, R. e Katayama, T. (2004). The Φ accrual failure detector. Em *23rd International Symposium on Reliable Distributed Systems (SRDS)*.

[Heimovski et al., 2017] Heimovski, G., Turchetti, R. C., Duarte, E. P., Wickboldt, J. A. e Granville, L. Z. (2017). Alta disponibilidade de um gerenciador de nuvem iaas baseada em replicação: Experiência & resultados. Em *XXXIV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'17)*.

[Ho et al., 2016] Ho, C. C., Wang, K. e Hsu, Y. H. (2016). A fast consensus algorithm for multiple controllers in software-defined networks. Em *18th International Conference on Advanced Communication Technology (ICACT)*.

[Hunt et al., 2010] Hunt, P., Konar, M., Junqueira, F. P. e Reed, B. (2010). ZooKeeper: Wait-free Coordination for Internet-scale Systems. Em *USENIX Conference on USENIX Annual Technical Conference (USENIXATC)*.

[Jacobson, 1988] Jacobson, V. (1988). Congestion Avoidance and Control. Em *Symposium Proceedings on Communications Architectures and Protocols*.

[Karakus e Durresi, 2017] Karakus, M. e Durresi, A. (2017). A survey: Control plane scalability issues and approaches in software-defined networking (SDN). *Computer Networks*, 112.

[Kebarighotbi e Cassandras, 2011] Kebarighotbi, A. e Cassandras, C. (2011). Timeout control in distributed systems using perturbation analysis. Em *50th IEEE Conference on Decision and Control and European Control Conference (CDC-ECC)*.

[Koponen et al., 2010] Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T. e Shenker, S. (2010). Onix: A Distributed Control Platform for Large-scale Production Networks. Em *9th Conference on Operating Systems Design and Implementation (OSDI)*.

[Kreutz et al., 2015] Kreutz, D., Ramos, F. M. V., Veríssimo, P. J. E., Rothenberg, C. E., Azodolmolky, S. e Uhlig, S. (2015). Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1).

[Lamport, 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.

[Lamport, 1998] Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2).

[Lamport, 2001] Lamport, L. (2001). Paxos Made Simple. *SIGACT News*, 32(4):51–58.

[Lamport, 2006a] Lamport, L. (2006a). Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125.

[Lamport, 2006b] Lamport, L. (2006b). Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2).

[Lara et al., 2014] Lara, A., Kolasani, A. e Ramamurthy, B. (2014). Network Innovation using OpenFlow: A Survey. *Communications Surveys Tutorials*, 16(1).

[Li et al., 2016] Li, J., Michael, E., Sharma, N. K., Szekeres, A. e Ports, D. R. K. (2016). Just say no to paxos overhead: Replacing consensus with network ordering. Em *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[Moraes e Duarte Jr., 2011] Moraes, D. M. e Duarte Jr., E. P. (2011). A failure detection service for internet-based multi-as distributed systems. Em *17th IEEE International Conference on Parallel and Distributed Systems ICPADS*.

[Ongaro e Ousterhout, 2014] Ongaro, D. e Ousterhout, J. (2014). In Search of an Understandable Consensus Algorithm. Em *USENIX Conference on USENIX Annual Technical Conference (USENIXATC)*.

[Openflow, 2015] Openflow (2015). Openflow 1.x discussion. `http://archive.openflow.org/wk/index.php/Openflow_1.X_Discussion`. Acessado em 23/07/2015.

[OPNFV, 2015] OPNFV (2015). An Open Plataform to Accelerate NFV. `https://www.opnfv.org/sites/opnfv/files/pages/files/opnfv_whitepaper_092914.pdf`. Acessado em 22/05/2017.

[Paxson et al., 2017] Paxson, V., Allman, M., Chu, H. J. e Sargent, M. (Acessado em 01/06/2017). Computing tcp's retransmission timer. `https://tools.ietf.org/html/rfc6298`.

[Pedone e Schiper, 2003] Pedone, F. e Schiper, A. (2003). Optimistic atomic broadcast: a pragmatic viewpoint. *Theoretical Computer Science (Elsevier)*, 291(1):79–101.

[Pham et al., 2015] Pham, C., Tran, H. D., Moon, S. I., Thar, K. e Hong, C. S. (2015). A General and Practical Consolidation Framework in CloudNFV. Em *International Conference on Information Networking (ICOIN)*.

[Postel, 1981] Postel, J. (1981). RFC 793: Transmission Control Protocol. Em *Request for Comments 793 (RFC793)*.

[Rashidi et al., 2017] Rashidi, B., Fung, C. e Bertino, E. (2017). A collaborative ddos defence framework using network function virtualization. *IEEE Transactions on Information Forensics and Security*, PP(99).

[Santos et al., 2011] Santos, N., Schiper, A., Hutle, M. e Borran, F. (2011). Quantitative analysis of consensus algorithms. *IEEE Transactions on Dependable and Secure Computing*, 9.

[Schiff et al., 2016] Schiff, L., Schmid, S. e Kuznetsov, P. (2016). In-Band Synchronization for Distributed SDN Control Planes. *SIGCOMM Comput. Commun. Rev.*, 46(1).

[Sergent et al., 2001] Sergent, N., Défago, X. e Schiper, A. (2001). Impact of a Failure Detection Mechanism on the Performance of Consensus. Em *Pacific Rim Symp. on Dependable Computing (PRDC)*.

[Sharma et al., 2011] Sharma, S., Staessens, D., Colle, D., Pickavet, M. e Demeester, P. (2011). Enabling fast failure recovery in OpenFlow networks. Em *8th International Workshop on the Design of Reliable Communication Networks (DRCN)*.

[Sherry et al., 2012] Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S. e Sekar, V. (2012). Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. Em *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*.

[Shumway e Stoffer, 2006] Shumway, R. H. e Stoffer, D. S. (2006). *Time Series Analysis and Its Applications With R Examples*. Springer.

[Souza et al., 2017] Souza, G. V., Turchetti, R. C., de Camargo, E. T. e Duarte, E. P. (2017). Uma função virtualizada de rede para a sincronização consistente do plano de controle em redes sdn. Em *XXXIV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'17)*.

[Tianzhu et al., 2016] Tianzhu, Z., Andrea, B., Samuele De, D. e Paolo, G. (2016). The role of inter-controller traffic for placement of distributed sdn controllers. Em *Networking and Internet Architecture.*

[Turchetti et al., 2016a] Turchetti, R. C., Duarte, E. P., Arantes, L. e Sens, P. (2016a). A QoS-configurable failure detection service for internet applications. *Journal of Internet Services and Applications*, 7(1).

[Turchetti et al., 2016b] Turchetti, R. C., Duarte, E. P., Arantes, L. e Sens, P. (2016b). Um serviço de detecção de falhas com qos auto-ajustável para múltiplas aplicações simultâneas. Em *XXXIV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'16)*.

[Turchetti e Duarte Jr., 2014] Turchetti, R. C. e Duarte Jr., E. P. (2014). Uma Nova Arquitetura para a Implementação de um Serviço de Detecção de Falhas na Internet. Em *I Workshop Pré-IETF. XXXIV Congresso da Sociedade Brasileira de Computação*.

[Turchetti e Duarte Jr., 2015a] Turchetti, R. C. e Duarte Jr., E. P. (2015a). Implementação de uma Função Virtualizada de Rede para Detecção de Falhas. Em *XVI Workshop de Testes e Tolerância a Falhas (WTF). XXXIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*.

[Turchetti e Duarte Jr., 2015b] Turchetti, R. C. e Duarte Jr., E. P. (2015b). Implementation of failure detector based on network function virtualization. Em *IEEE International Conference on Dependable Systems and Networks Workshops, DSN Workshops 2015, Rio de Janeiro, Brazil, June 22-25, 2015*.

[Turchetti e Duarte Jr., 2016] Turchetti, R. C. e Duarte Jr., E. P. (2016). Computing a precise timeout interval despite a communication channel with varying behavior. Em *Proceedings of the 2016 workshop on Fostering Latin-American Research in Data Communication Networks, LANCOMM@SIGCOMM 2016*.

[Turchetti e Duarte Jr., 2017] Turchetti, R. C. e Duarte Jr., E. P. (2017). Nfv-fd: Implementation of a failure detector using network virtualization technology. *Int. Journal of Network Management*, 27(0).

[van Adrichem et al., 2014] van Adrichem, N. L. M., van Asten, B. J. e Kuipers, F. A. (2014). Fast Recovery in Software-Defined Networks. Em *Third European Workshop on Software Defined Networks, EWSDN*.

[Vilalta et al., 2015] Vilalta, R., Munoz, R., Mayoral, A., Casellas, R., Martinez, R., Lopez, V. e Lopez, D. (2015). Transport network function virtualization. *Journal of Lightwave Technology*.

[Wiesmann et al., 2006] Wiesmann, M., Urbán, P. e Défago, X. (2006). An SNMP based failure detection service. Em *25th IEEE Symposium on Reliable Distributed Systems (SRDS'06)*.

[Xilouris et al., 2014] Xilouris, G., Trouva, E., Lobillo, F., Soares, J., Carapinha, J., McGrath, M., Gardikis, G., Paglierani, P., Pallis, E., Zuccaro, L., Rebahi, Y. e Kourtis, A. (2014). T-NOVA: A marketplace for virtualized network functions. Em *European Conference on Networks and Communications (EuCNC)*.

[Zhou et al., 2014] Zhou, B., Wu, C., Hong, X. e Jiang, M. (2014). Programming network via distributed control in software-defined networks. Em *2014 IEEE International Conference on Communications (ICC)*.

[Zia et al., 2009] Zia, H. A., Sridhar, N. e Sastry, S. (2009). Failure Detectors for Wireless Sensor-Actuator Systems. *Ad Hoc Networks*, 7(5).

Apêndice A

Publicações

A.1 Trabalhos Publicados no Âmbito da Tese

1. Souza, G. V., **Turchetti, R. C.**, de Camargo, E. T. e Duarte, E. P. A Virtual Network Function for Maintaining a Consistent SDN Control Plane. IEEE International Conference on Computer Communications (INFOCOM'18). (submitted).

2. [Souza et al., 2017] Souza, G. V., **Turchetti, R. C.**, de Camargo, E. T. e Duarte, E. P. (2017). Uma função virtualizada de rede para a sincronização consistente do plano de controle em redes sdn. Em XXXIV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'17).

3. [Turchetti et al., 2016b] **Turchetti, R. C.**, Duarte, E. P., Arantes, L. e Sens, P. (2016b). Um serviço de detecção de falhas com qos auto-ajustável para múltiplas aplicações simultâneas. Em XXXIV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'16).

4. [Turchetti e Duarte Jr., 2016] **Turchetti, R. C.** e Duarte Jr., E. P. (2016). Computing a precise timeout interval despite a communication channel with varying behavior. Em Proceedings of the 2016 workshop on Fostering Latin-American Research in Data Communication Networks, LANCOMM@SIGCOMM 2016.

5. [Turchetti e Duarte Jr., 2015b] **Turchetti, R. C.** e Duarte Jr., E. P. (2015b). Implementation of failure detector based on network function virtualization. Em IEEE International Conference on Dependable Systems and Networks Workshops, DSN Workshops 2015, Rio de Janeiro, Brazil, June 22-25, 2015.

6. [Turchetti e Duarte Jr., 2015a] **Turchetti, R. C.** e Duarte Jr., E. P. (2015a). Implementação de uma Função Virtualizada de Rede para Detecção de Falhas. Em XVI Workshop de Testes e Tolerância a Falhas (WTF). XXXIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos.

7. [Turchetti e Duarte Jr., 2014] **Turchetti, R. C.** e Duarte Jr., E. P. (2014). Uma Nova Arquitetura para a Implementação de um Serviço de Detecção de Falhas na Internet. Em I Workshop Pré-IETF. XXXIV Congresso da Sociedade Brasileira de Computação.

Trabalhos em Periódicos:

1. [Turchetti et al., 2016a]**Turchetti, R. C.**, Duarte, E. P., Arantes, L. e Sens, P. (2016a). A QoS- configurable failure detection service for internet applications. Em *Journal of Internet Services and Applications*, 7(1).

2. [Turchetti e Duarte Jr., 2017]**Turchetti, R. C.** e Duarte Jr., E. P. (2017). NFV-FD: Implementation of a failure detector using network virtualization technology. Em *Journal of Network Management*, 27(0).

A.2 Trabalho Publicado no Âmbito do Grupo de Pesquisa

1. [Heimovski et al., 2017] Heimovski, G., **Turchetti, R. C.**, Duarte, E. P., Wickboldt, J. A. e Granville, L. Z. (2017). Alta disponibilidade de um gerenciador de nuvem iaas baseada em replicação: Experiência & resultados. Em XXXIV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'17).